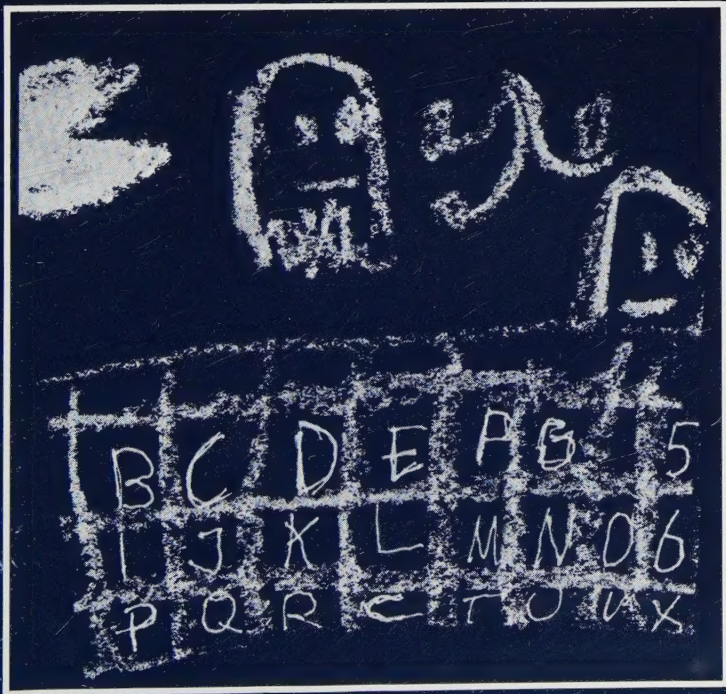
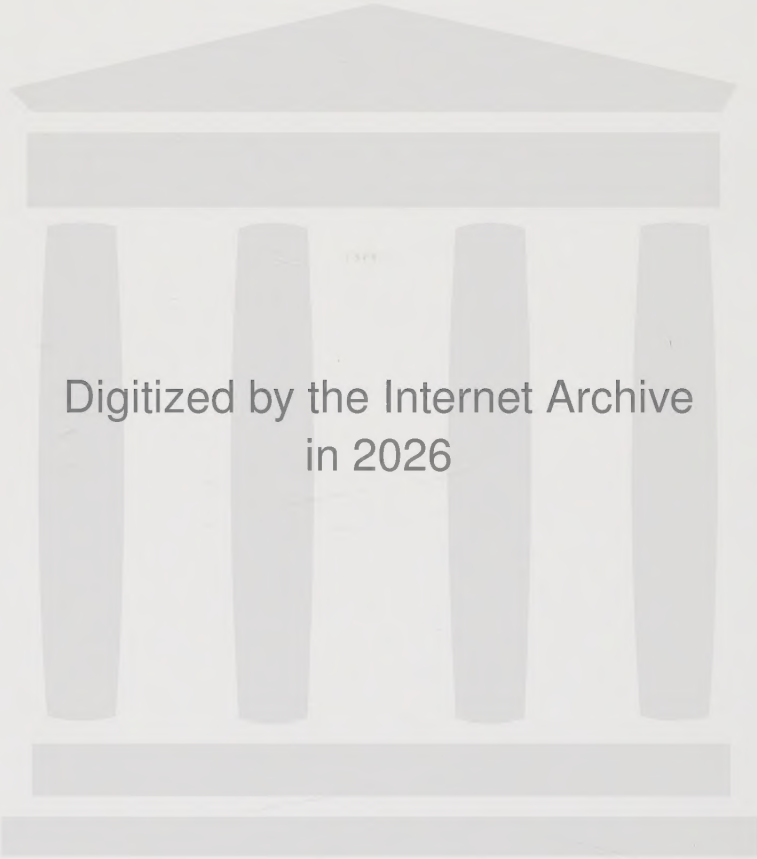


Revised 1983

AN ASSOCIATIVE ARRAY PROCESSOR SUPPORTING A RELATIONAL ALGEBRA

Ivan Kruzela





Digitized by the Internet Archive
in 2026

https://archive.org/details/IA41552407_0054

An Associative Array Processor Supporting A Relational Algebra

AKADEMISK AVHANDLING

som för avläggande av teknisk doktorsexamen vid tekniska fakulteten vid Universitetet i Lund kommer att offentligen försvaras i hörsal E:B, Lunds Tekniska Högskola, fredagen den 20 maj 1983 kl 10.15

av

Ivan Kruzela

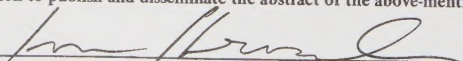
civ.ing.

Organization LUND UNIVERSITY Avd. för Digital- och Datorteknik, LTH Box 725, 220 07 LUND, Tel 046/107515 Dept. of Computer Engineering, Univ. of Lund, PO Box 725, S-220 07 LUND, Sweden		Document name DOCTORAL DISSERTATION	
Author(s) Kruzela, Ivan		Date of issue May 1983	
		CODEN: LUTEDX/(TEDD-1003)/1-253/(1983)	
		Sponsoring organization	
Title and subtitle An Associative Array Processor Supporting A Relational Algebra.			
Abstract The research presented in this thesis deals with exploring the possibilities offered by an Associative Array Processor in the area of a relational database management. The feasibility of the Associative Array as a component of a special purpose database computer is investigated. The starting-point for this research is a highly parallel computer LUCAS (Lund University Content Addressable System), which has been designed and implemented at the University of Lund. LUCAS consists of 128 bit-serial processors arranged in a linear array, and working in a SIMD mode. LUCAS is assumed to be a model of an Associative Array. In the thesis a demonstration of implementation of basis operations of relational algebra, e.g. Join, Projection, and Semi-join, is given and their efficiency is discussed. Furthermore, a method of evaluating complex queries to a database stored in the Associative Array is shown. The performance of a database computer built with the Associative Array is compared with the performance of other designs. Finally, the use of the Associative Array for evaluation of the Join operation on very large realations is discussed.			
Key words associative memory, associative processor, database management, database machine, database computer, relational algebra, LUCAS, join processor.			
Classification system and/or index terms (if any) Computer Science, Electrical Engineering			
Supplementary bibliographical information		Language English	
ISSN and key title		ISBN	
Recipient's notes	Number of pages 253	Price	
	Security classification		

Distribution by (name and address)

I, the undersigned, being the copyright owner of the abstract of the above-mentioned dissertation, hereby grant to all reference sources permission to publish and disseminate the abstract of the above-mentioned dissertation.

Signature



Date

April 19, 1983

Department of Computer Engineering
University of Lund S-220 07 Lund, Sweden

AN ASSOCIATIVE ARRAY PROCESSOR SUPPORTING A RELATIONAL ALGEBRA

Ivan Kruzela



AN ASSOCIATIVE ARRAY
PROCESSOR SUPPORTING
A RELATIONAL ALGEBRA

WERNER KUNZ



Printed in Sweden
Studentlitteratur
Lund 1983

Chapter 1. Introduction	1
1.1 The Problem	1
1.2 The Objectives	2
1.3 The Scope	3
1.4 The Methodology	4
Chapter 2. Literature Review	5
2.1 Theoretical Framework	5
2.2 Empirical Studies	6
2.3 Theoretical Framework	7
2.4 Empirical Studies	8
2.5 Theoretical Framework	9
2.6 Empirical Studies	10
2.7 Theoretical Framework	11
2.8 Empirical Studies	12
2.9 Theoretical Framework	13
2.10 Empirical Studies	14
2.11 Theoretical Framework	15
2.12 Empirical Studies	16
2.13 Theoretical Framework	17
2.14 Empirical Studies	18
2.15 Theoretical Framework	19
2.16 Empirical Studies	20
2.17 Theoretical Framework	21
2.18 Empirical Studies	22
2.19 Theoretical Framework	23
2.20 Empirical Studies	24
2.21 Theoretical Framework	25
2.22 Empirical Studies	26
2.23 Theoretical Framework	27
2.24 Empirical Studies	28
2.25 Theoretical Framework	29
2.26 Empirical Studies	30
2.27 Theoretical Framework	31
2.28 Empirical Studies	32
2.29 Theoretical Framework	33
2.30 Empirical Studies	34
2.31 Theoretical Framework	35
2.32 Empirical Studies	36
2.33 Theoretical Framework	37
2.34 Empirical Studies	38
2.35 Theoretical Framework	39
2.36 Empirical Studies	40
2.37 Theoretical Framework	41
2.38 Empirical Studies	42
2.39 Theoretical Framework	43
2.40 Empirical Studies	44
2.41 Theoretical Framework	45
2.42 Empirical Studies	46
2.43 Theoretical Framework	47
2.44 Empirical Studies	48
2.45 Theoretical Framework	49
2.46 Empirical Studies	50
2.47 Theoretical Framework	51
2.48 Empirical Studies	52
2.49 Theoretical Framework	53
2.50 Empirical Studies	54
2.51 Theoretical Framework	55
2.52 Empirical Studies	56
2.53 Theoretical Framework	57
2.54 Empirical Studies	58
2.55 Theoretical Framework	59
2.56 Empirical Studies	60
2.57 Theoretical Framework	61
2.58 Empirical Studies	62
2.59 Theoretical Framework	63
2.60 Empirical Studies	64
2.61 Theoretical Framework	65
2.62 Empirical Studies	66
2.63 Theoretical Framework	67
2.64 Empirical Studies	68
2.65 Theoretical Framework	69
2.66 Empirical Studies	70
2.67 Theoretical Framework	71
2.68 Empirical Studies	72
2.69 Theoretical Framework	73
2.70 Empirical Studies	74
2.71 Theoretical Framework	75
2.72 Empirical Studies	76
2.73 Theoretical Framework	77
2.74 Empirical Studies	78
2.75 Theoretical Framework	79
2.76 Empirical Studies	80
2.77 Theoretical Framework	81
2.78 Empirical Studies	82
2.79 Theoretical Framework	83
2.80 Empirical Studies	84
2.81 Theoretical Framework	85
2.82 Empirical Studies	86
2.83 Theoretical Framework	87
2.84 Empirical Studies	88
2.85 Theoretical Framework	89
2.86 Empirical Studies	90
2.87 Theoretical Framework	91
2.88 Empirical Studies	92
2.89 Theoretical Framework	93
2.90 Empirical Studies	94
2.91 Theoretical Framework	95
2.92 Empirical Studies	96
2.93 Theoretical Framework	97
2.94 Empirical Studies	98
2.95 Theoretical Framework	99
2.96 Empirical Studies	100
2.97 Theoretical Framework	101
2.98 Empirical Studies	102
2.99 Theoretical Framework	103
2.100 Empirical Studies	104

To Pavla, Petra and Filip

CONTENTS

<u>Chapter 1. INTRODUCTION</u>	9
1.1 MOTIVATION FOR THE RESEARCH	9
1.2 LUCAS	11
1.3 ORGANIZATION OF THE THESIS	13
1.4 ACKNOWLEDGEMENTS	14
<u>Chapter 2. LUCAS ASSOCIATIVE COMPUTER</u>	15
2.1 SYSTEM OVERVIEW	15
2.2 CONTROL UNIT	21
2.2.1 Microprogram Control Part	22
2.2.2 Address Processor	29
2.2.3 Comparand and Mask Registers	35
2.2.4 Interface	37
2.3 ASSOCIATIVE ARRAY	40
2.3.1 Overview	40
2.3.2 Processor in the Associative Array	44
2.4 SYSTEM OPERATION	51
2.4.1 Program Execution	51
2.4.2 Data I/O	53
2.5 PHYSICAL DESCRIPTION	55
<u>Chapter 3. DATABASE SYSTEMS</u>	57
3.1 BASIC CONCEPTS	57
3.2 DATA MODELS	62
3.2.1 Hierarchical Data Model	63
3.2.2 Network Data Model	64
3.2.3 Relational Data Model	65
3.2.4 Comparison of the three Models	66
3.3 RELATIONAL DATABASE	67
3.3.1 Relations	67
3.3.2 Data Manipulation Languages	69
3.3.3 Relational Calculus	70
3.3.4 Relational Algebra	70

<u>Chapter 4. DATABASE COMPUTERS</u>	80
4.1 CATEGORIZATION OF DATABASE COMPUTERS	80
4.2 CELLULAR LOGIC SYSTEMS	83
4.3 BACKEND COMPUTERS	89
4.4 INTEGRATED DATABASE MACHINES	94
4.5 HIGH-SPEED ASSOCIATIVE ARRAY SYSTEMS	96
<u>Chapter 5. RELATIONAL ALGEBRA ON LUCAS</u>	105
5.1 REPRESENTATION OF A RELATION IN THE ASSOCIATIVE ARRAY	108
5.2 BASIC OPERATIONS IN THE ASSOCIATIVE ARRAY	111
5.2.1 Bitslice Operations	111
5.2.2 Byte Transfer Operations	118
5.2.3 Word Transfer Operations	126
5.2.4 Compare Operations	130
5.3 INTERNAL ALGORITHMS FOR ALGEBRAIC OPERATIONS	132
5.3.1 Algorithms creating a Markbitslice of the Result	133
5.3.2 Algorithms that assemble the Result	149
5.4 DISCUSSION	158
5.4.1 Timing Equations	158
5.4.2 Performance Evaluation	162
5.4.3 Comparison with Alternative Designs	172
<u>Chapter 6. INTERNAL QUERY EVALUATION</u>	175
6.1 A TYPICAL DATABASE AND A SAMPLE OF QUERIES	177
6.2 EVALUATION OF THE QUERIES IN THE ASSOCIATIVE ARRAY	182
6.3 DISCUSSION	208

<u>Chapter 7. COMPARATIVE PERFORMANCE EVALUATION OF A DATABASE COMPUTER</u>	212
7.1 SPECIFICATION OF CHARACTERISTIC OF DATABASE MACHINES	213
7.2 DATABASE AND QUERIES	216
7.3 LUCAS RESPONSE TIMES	219
7.4 PERFORMANCE COMPARISONS	221
7.5 INFLUENCE OF THE SIZE OF ASSOCIATIVE ARRAY	224
7.6 CONCLUSIONS	226
 <u>Chapter 8. EXTERNAL EVALUATION OF JOINS</u>	 228
8.1 SYSTEM DESCRIPTION	230
8.2 ALGORITHM AND TIMING EQUATIONS	232
8.3 DISCUSSION	236
 <u>Chapter 9. CONCLUSIONS AND FUTURE RESEARCH</u>	 239
9.1 CONCLUSIONS	240
9.2 VLSI IMPLEMENTATION	243
 <u>REFERENCES</u>	 249

Chapter 1.

INTRODUCTION

1.1 MOTIVATION FOR THE RESEARCH

Though today it is familiar to most people, in some intuitive but mostly correct interpretation, the term database appeared in the literature about information processing for the first time as late as in 1964 [McGee81]. A database is stored in the memory of a computer, and to handle it a new type of software, a database management system, DBMS, evolved. The practical need for more efficient systems for managing the information in the database soon gave rise to new methodologies, new programming languages, new algorithms and also new hardware techniques. A new field of human enterprise, database technology, emerged and its importance is still growing.

The software of present database management systems is very complex and expensive and its efficiency is not always adequate to the needs of its users. The reason is that a DBMS is usually implemented on a conventional general-purpose computer which was designed for other kinds of applications.

The von Neumann computer model, developed in the mid 1940s, was intended to be employed in numerical applications, where basic operations are addition, subtraction, multiplication, etc, and basic data types are numbers stored in a memory and addressed by locations. The proper use of this type of computer is sequential

calculations in loops.

The purpose of a DBMS is not calculation but rather manipulation of large volumes of data. Basic operations required are retrieval and updating of data, basic data types are records which are identified by contents rather than by locations. None of those features are supported by the hardware of general-purpose computers. Furthermore, the DBMS offers a great natural potential for parallel execution which is impossible to exploit in a conventional computer.

This disharmony between means and goals became more and more apparent in the late 1960s as databases grew larger, and more sophisticated functions were being incorporated into DBMSs to satisfy growing user demands. In the early 1970s, people at a number of universities in the USA initiated pioneering research projects in the area of special purpose computers for data management. Since then, this area has become one of the most dynamic research fields in the domain of computer architecture. In an ever increasing stream, numerous papers are published each year, dealing with description, analysis and discussion of new designs and new concepts of database computers.

There are a number of ways to organize the information in a database. One particularly important approach is the logical organization of the data in the form of tables called relations. This approach has many advantages, some of which will be discussed in Chapter 3.

Its main disadvantage is commonly taken to be the fact that a table is a two-dimensional structure and therefore must be translated into a one dimensional string of data if it is to be sequentially processed in a conventional computer. This implies a need for an elaborate software interface between the logical data model seen by the user and the physical storage structure. Even if the relational database management system can be implemented efficiently this necessary interface is responsible for a costly overhead and for the large complexity of the system.

This disadvantage, however, can be turned into an advantage since the simplicity of the two-dimensional table gives an opportunity to exploit new forms of hardware organizations. The most natural way to store and to process tables would be in a computer structure which also looks like a table and where a one-to-one correspondence between the logical and the physical data organization can be achieved. Furthermore, a table containing data is a unity, and the natural way to process it in this table-like hardware structure would be in parallel, by operations having tables as operands.

Fortunately, such a computer structure, which can make the management of a relational database simple and efficient does exist. It has been known for a long time, but its potential has never been fully explored. In the literature it has many names. We will call it an Associative Processor or an Associative Array.

The research presented in this thesis deals with exploring the possibilities offered by the Associative Array in the area of a relational database management. The starting-point for this research is the Associative Array processor LUCAS (Lund University Content Addressable System), which has been designed and implemented at the Department of Computer Engineering, University of Lund. Our conclusion is that an Associative Array modelled on LUCAS can indeed be more useful than previously thought.

1.2 LUCAS

The work on the project was begun in 1978, by a research group of three persons. The main inspiration for the research work was the monograph 'Content Addressable Processors' by Caxton Foster [Foster77]. The goal of the project was to acquire basic knowledge and experience in the promising area of associative processing by building a working associative processor, and to perform evaluation tests in selected application areas. LUCAS is a research vehicle, and one of the main objectives was to make it flexible to provide for

the testing of different architectural principles.

The development towards the implementation of LUCAS proceeded in four steps: literature study, simulation, construction of a simple prototype, and the final design of LUCAS.

The first step was the study of available literature about associative computers. Surveys on this topic were given by Thurber [Thurber75, Thurber76], Yau and Feng [Yau77], and Pahrami [Pahrami73]. The interest in associative computing has been steadily increasing during the last two decades. Numerous proposals for the design of associative computers can be found, yet only a few real machines have been built and actually used. There are many application areas for the associative computers, but so far the factor limiting their wide use has been the lack of cost efficiency. The associative computer is more complex than the ordinary von Neumann computer and consequently more expensive. This situation is now changing, due to the tremendous development in the field of semiconductor technology. Since the associative computer has a regular structure it is very suitable for implementation in VLSI. In the near future, it should be possible to build large powerful associative computers at an acceptable total cost.

The second step was the preliminary design of a bit-serial word-parallel associative computer. This machine was simulated and its performance evaluated. The input to the simulator program was microprograms for different operations, the output was snapshots of the state of the Associative Array during execution. The main benefit of the simulation study was that different concepts could be conveniently tested before they were included in the final design.

The first prototype consisted of a microprogrammable control unit and an associative array with 8 processors. This prototype was evaluated in simple situations.

The decision to build a full scale associative computer with 128 processors in the associative array was taken in the summer of 1980. In the spring of 1981, a system with 8 processors was running.

During 1981, new processors were successively added until the full system with 128 processors was finished in the spring of 1982.

1.3 ORGANIZATION OF THE THESIS

The thesis is organized as follows:

Chapter 2 describes the implementation of the LUCAS system. Hopefully, this description is detailed enough to be a source of inspiration, or a warning example, to other people who intend to design similar computers.

Chapter 3 is a brief survey of the field of database management. The emphasis lies on the relational data model. It also gives, by way of simple examples, informal definitions of operations of relational algebra.

In Chapter 4 we review a sample of database computers, some of which will be discussed in later chapters.

Chapter 5 gives a description of the implementation of relational algebra operations on LUCAS, when it is assumed that the size of the operand relations is such that they can be stored in the Associative Array. Furthermore, this chapter contains a discussion of the performance of very large Associative Arrays.

Chapter 6 goes a step beyond the material in Chapter 5. We show a simple but powerful method of evaluating queries to a database stored in the Associative Array.

Chapter 7 is a discussion of the performance of a database computer built with an Associative Array.

Chapter 8 studies the usefulness of an Associative Array for evaluation of the Join operation on very large relations.

In Chapter 9, finally, we suggest some topics for future research and discuss the possibility of a VLSI implementation of a large Associative Array.

1.4. ACKNOWLEDGEMENTS

I would like to express my gratitude to Rolf Johannesson for his encouragement and his constant friendship during the course of this research.

I am deeply indebted to my colleagues Christer Fernström and Bertil Svensson, together with whom I have carried out the work and without whom LUCAS would never exist.

I would like to thank Anders Ardö for his text formatting program SubScribe Ver. 1.4, Lena Månsson and Kerstin Johnson for excellent drawings, Josef Wajnbloom and Dan Leek for technical assistance, and Björn Breidegaard for making us believe in our technical competence in times when nothing worked.

Many people gave me their help and encouragement. Thank you, Göran, Hans, Laila, Lars, Lennart, Mats, Peter, Pär, Staffan, and Olle.

Chapter 2.

LUCAS ASSOCIATIVE COMPUTER

2.1 SYSTEM OVERVIEW

The LUCAS system, shown in Figure 2.1, consists of three parts: a Supporting Processor (sometimes called the host) , a Control Unit and an Associative Array.

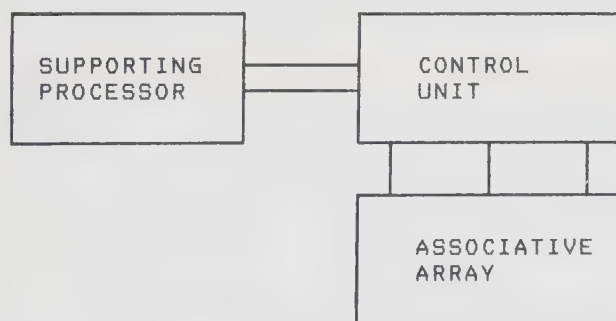


Fig. 2.1 LUCAS

The Supporting Processor handles input and output of data to and from the Associative Array, controls execution of programs on data in the Associative Array and interacts with the user. Furthermore, the Supporting Processor is used for editing, assembling and compiling of programs and microprograms.

The Supporting Processor is a commercial PROLOG Z-80 8-bit microcomputer with a CP/M operating system consisting of the following cards:

- * A Z-80 CPU card.
- * Two memory cards.
- * A floppy disk controller card.
- * A peripheral interface card for communication with a terminal and printer.
- * A card with buffers, interconnecting the Supporting Processor and the Control Unit

The cards, communicating via a standard 56-pin STD bus, are plugged into a motherboard rack with 16 slots. The main memory of the system consists of 63 kbytes RAM. The secondary memory is a dual floppy-disk system with a capacity of 250 kbytes/disk. Via a 9600 baud serial link the Supporting Processor is also connected to a VAX 11/780.

The choice of Supporting Processor is not essential. Any among many commercial microcomputer or minicomputer systems would do as well. The Z-80 microcomputer system was chosen because of its availability at the initial state of the project.

The Control Unit, which is microprogrammable, interprets instructions received from the Supporting Processor and sends microinstructions to the Associative Array. The Control Unit includes a Microprogram Controller sequencing microprograms, a Microprogram Memory of 4096 80-bit words, an 80-bit wide Pipeline Register, and a fast Address Processor computing addresses to the Associative Array. All internal registers of the Control Unit which interact with the Supporting Processor, e.g. the Instruction Register, the Status Register, and the Parameter Registers with parameters for microprograms, are mapped to its address space. This feature makes the interface between the Control Unit and the

Supporting Processor, both at the hardware and software level, very simple and convenient.

The Associative Array consists of 128 simultaneously operating processors with a bit serial working mode. In each clock cycle, all 128 processors execute the same microinstruction. A processor includes a Processing Element with four flip-flops for storing intermediate results, a 4096-bit memory word with data, and an 8-bit I/O Register for input and output of data. The processors in the Array can exchange information through an interconnection network or with the help of the I/O Registers.

One of the four flip-flops in the Processing Element, called the Tag, can prohibit writing of data to its associated memory word. The Tags are the hardware source of associativity of data processing in the Associative Array. Though all 128 processors always execute the same instruction, data are updated only in memory words of processors which have their Tag set to one. Usually, the content of the Tags is the result of some previous search operation selecting memory words with data of some desired property for further processing.

The size of the Associative Array (128 processors) could easily be extended if there was a need for improving performance.

There are two alternative ways to look at the Associative Array. The first, used in the description above, is to see it as a linear array of identical processors. The second is to see it as consisting of a Memory Array of 128 words, 4096 bits wide where to each word is connected a simple processor and an I/O shift register. Which description is preferred depends on the reader.

In addition to the Tags, LUCAS also provides two other typical components of an associative computer: a Comparand Register and a Mask Register. Logically, they are parts of the Associative Array, but, for reasons of uniformity of the Array, they are physically implemented in the Control Unit.

The operation on data in the Associative Array, directed by the Control Unit, is mediated by execution of horizontally organized microprograms. A LUCAS microprogram consists of a sequence of 80-bit wide microinstruction words. All actions in the Control Unit and the Associative Array are determined by values in separate fields of the current microinstruction word in the Pipeline Register. At the end of each clock cycle, the next microinstruction word is loaded into the Pipeline Register from the Microprogram Memory. The address is supplied by the Microprogram Controller. The disposition of control fields of the microinstruction word is shown in Figure 2.2. We will refer to the control fields in the next two sections, since it will simplify explanation of details of the implementation of LUCAS.

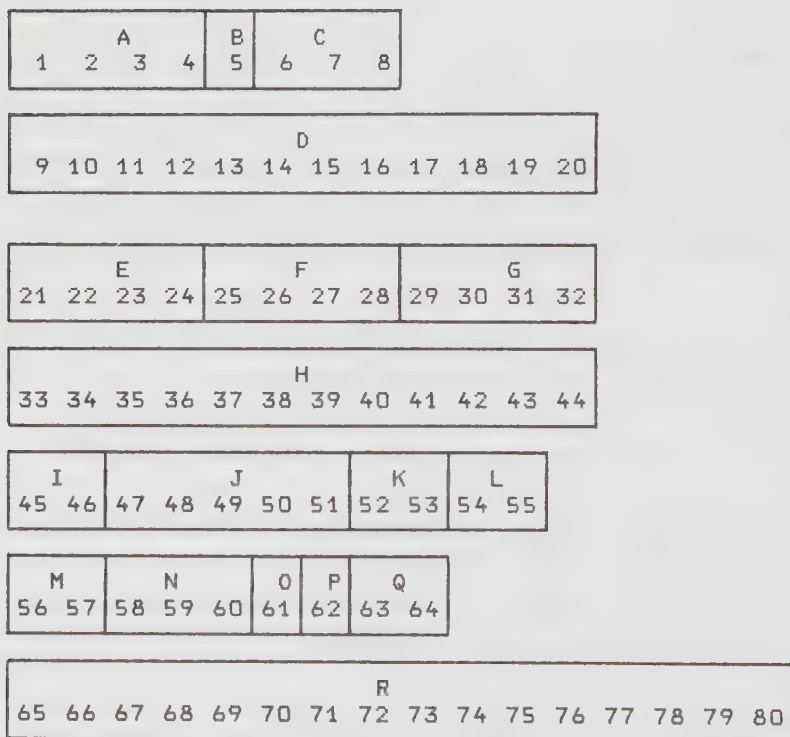


Fig. 2.2 Microinstruction word

The fields of the microinstruction word are the following:

- * A (bits 1-4): The Microprogram Controller instruction. There are 16 instructions sequencing the flow of the microprogram.
- * B (5): The polarity of a signal tested by the Microprogram Controller when it is executing conditional instructions.
- * C (6-8): The control of the Test Selector for the Microprogram Controller. The status of a number of test conditions in the Control Unit or in the Associative Array can influence the flow of the microprogram.
- * D (9-20): This field contains a twelve bit data value used as input to the Microprogram Controller. Usually, it is a destination address of a jump in a microprogram or an address to a subroutine. The value in this field can also be a constant determining how many times a loop in a microprogram will be executed.
- * E (21-24): The instruction to the Arithmetic Unit in the Address Processor. There are 16 operations that the Arithmetic Unit can perform, e.g. integer addition, incrementing and decrementing of an internal register, etc.
- * F (25-28): The A-address to the Arithmetic Unit in the Address Processor.
- * G (29-32): The B-address to the Arithmetic Unit in the Address Processor. There are 16 internal registers in the Arithmetic Unit holding addresses to the Associative Array. The A- and B- addresses point them out.
- * H (33-44): This field contains a twelve bit data value to be used as input to the Address Processor. This is usually a constant or a fixed address to data in the Associative Array.

- * I (45-46): This field controls the Address Stack in the Address Processor which enlarges the available space for holding addresses to the Associative Array.
- * J (47-51): This field is left for use in future expansion of LUCAS.
- * K (52-53): Control of the Loop-counter Register in the Address Processor used for implementing loops in microprograms.
- * L (54-55): Control of writing to the Comparand Register and to the Mask Register.
- * M (56-57): Control of the I/O Buffer Register which is used for communication between processors in the Associative Array and the Control Unit.
- * N (58-60): Control of the source of data for the Address Processor. Parameters necessary for computing addresses to data in the Associative Array can originate from a number of different registers in the Control Unit.
- * O (61): Control of the Busy flip-flop. This flip-flop is used for synchronization of the operation of the Control Unit and the Supporting Processor.
- * P (62): Control of the Status Register. Various test conditions can be loaded into the Status Register and later tested by the Supporting Processor.
- * Q (63-64): This field is used for test purposes during debugging of microprograms.
- * R (65-80): This field controls the operation of the Associative Array.

A detailed description of the functions controlled by field R will be given in Section 2.3.

2.2 CONTROL UNIT

The Control Unit governs the operation of the Associative Array. The organization of the Control Unit is shown in Figure 2.3. The Control Unit can be logically divided into four parts:

- * A Microprogram Control Unit controlling the flow of microprograms and generating all control signals.
- * An Address Processor, computing addresses to data in the Associative Array.
- * A Comparand Register and a Mask Register.
- * An Interface, providing for data, address, and control signal transfer between the Supporting Processor, the Control Unit and the Associative Array.

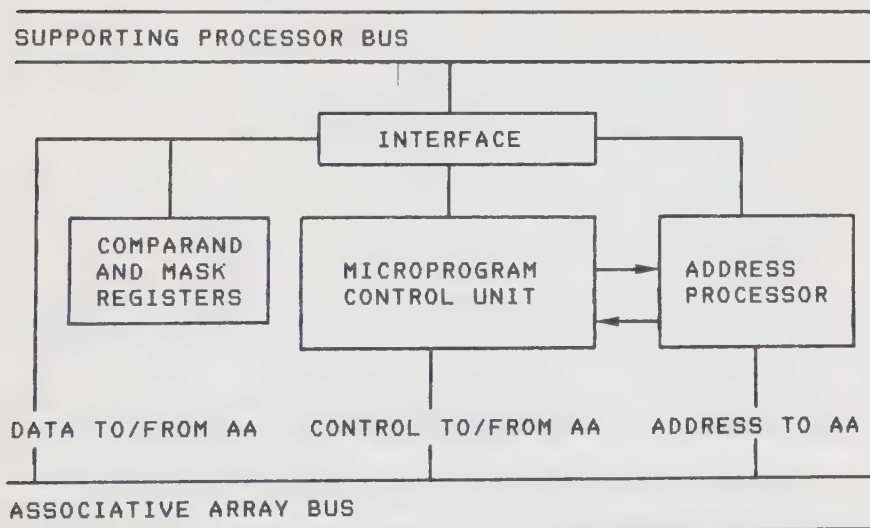


Fig. 2.3 Control Unit

2.2.1 MICROPROGRAM CONTROL UNIT

The Microprogram Control Unit, shown in Figure 2.4, is the heart of the Control Unit.

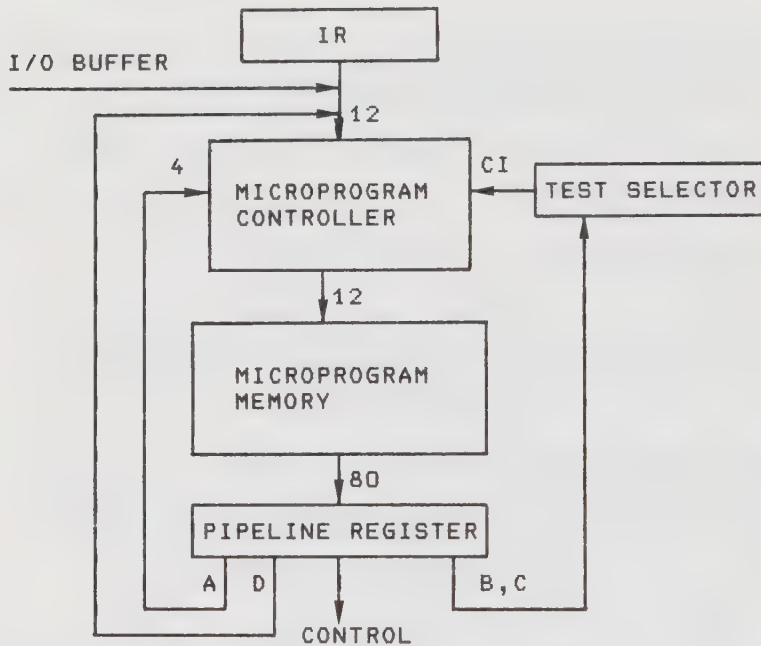


Fig. 2.4 Microprogram Control Unit

The Microprogram Control Unit consists of the following five units:

- * An 8-bit Instruction Register (IR), receiving instructions from the Supporting Processor.
- * A Microprogram Controller, generating addresses to the Microprogram Memory.
- * A Microprogram Memory of 4096 words, each 80-bits wide, storing the microprograms.
- * An 80-bit wide Pipeline Register, receiving a microinstruction word from the Microprogram Memory and

distributing control signals to the Control Unit and the Associative Array.

- * A Test Selector for the Microprogram Controller, selecting the condition to be tested by the conditional instructions of the Microprogram Controller.

An instruction code in the Instruction Register, loaded by the Supporting Processor, is used as the start address to a microprogram stored in the Microprogram Memory.

The flow of the microprogram is controlled by the Am2910 Microprogram Controller component which can execute 16 different sequencing instructions [Mick80]. It generate 12-bit addresses to the Microprogram Memory. A somewhat simplified picture of the internal organization of Am2910 is given in Figure 2.5.

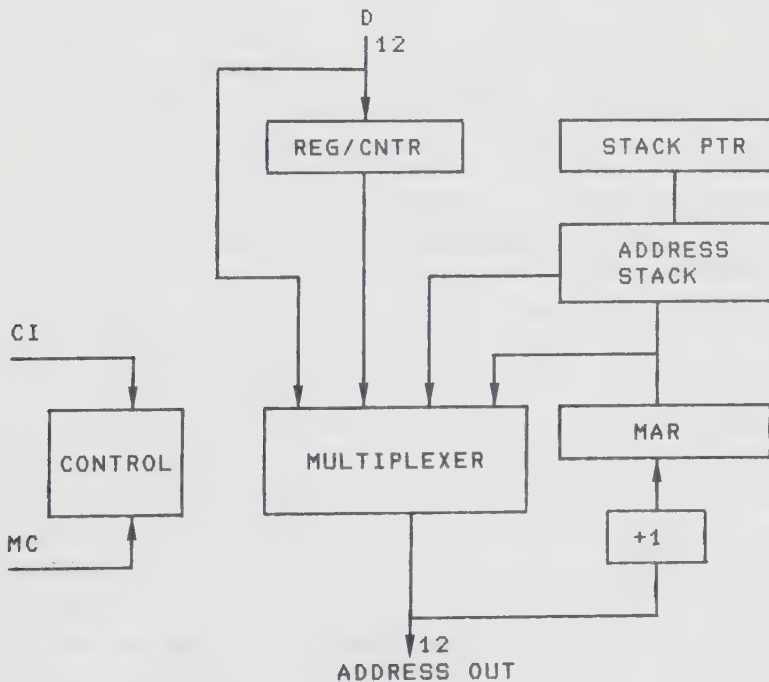


Fig. 2.5 Sequencer

The function of the Microprogram Controller is determined by three sets of inputs:

- * A 12-bit Direct Data Input (D).
- * A 4-bit microinstruction code (MC). The source of the microinstruction code is field A of the microinstruction word.
- * A 1-bit Condition Input (CI in Figure 2.4 and 2.5). The status of this input is tested by conditional microinstructions.

Data at the Direct Data Input may originate from one of three sources:

- * Field D (D in Figure 2.4) of the microinstruction word, containing the destination of jumps and subroutine calls or a loop-counter value.
- * Instruction Register with the code of an instruction. When this 8-bit register is used as the source of data, the four most significant bits at the 12-bit direct input are set to zero. A consequence of this hardware feature is that microprograms of all instructions must start in the first 256 words of the Microprogram Memory, which hold a jump table.
- * I/O Buffer Register. This rather exotic feature makes it possible to use data in the Associative Array as addresses to microprograms (instructions) or as loop-counter values. If the data are loaded from the 8-bit I/O Buffer Register, the most significant 4 bits are set to one.

The source of the signal at the condition input of the Microprogram Controller is the Test Selector which is implemented with a multiplexer. The Test Selector is controlled by field C of the microinstruction word. The following signals connected to the inputs

of the multiplexer can be gated and tested by conditional instructions in the Microprogram Controller:

- * ZAP: A signal indicating whether the current microoperation in the Arithmetic Unit of the Address Processor gives the result zero or not. A typical use of this signal is for testing loop terminating conditions.
- * MASK: The value of the bit in the Mask Register at the current address from the Address Processor. This signal indicates to the Microprogram Controller, for example, that a bitslice at the current address in the Associative Array should not be processed.
- * ZLC: A signal indicating that the Loop-counter Register in the Address Processor contains the value zero. The Loop-counter Register is used for supporting loops in microprograms.
- * BUSY: An output of the Busy flip-flop indicating whether the Supporting Processor has loaded a new instruction in the Instruction Register or not.
- * NONE: A signal from the Associative Array indicating if any of the Tag flip-flops are set to one.
- * ZERO: A logical zero. This signal is used for making conditional instructions unconditional.

The polarity of the signal at the Condition Input is determined by field B of the microinstruction word. The Microprogram Controller may test either the signals described above or their complements.

During each microinstruction, the Microprogram Controller generates a 12-bit address to the Microprogram Memory. This address may come from one of four sources (compare Figure 2.5):

- * A Microprogram Address Register (MAR). This register contains an address one greater than the previous one.

- * A Direct Data Input with a value from the Pipeline Register, the Instruction Register or the I/O Buffer Register.
- * A Register/Counter (REG/CNTR) with data loaded from the Direct Data Input during some previous instruction. This register is used for holding the address to the Microprogram Memory or for counting loops in microprograms.
- * The top of a five level address stack holding return addresses from subroutines and starting addresses of loops.

The Microprogram Memory can work in two modes :

- * A run mode, when it is logically organized as 4096 words, each 80 bits wide.
- * A load mode, when it is logically organized as 40960 words, each 8 bits wide.

The mode of operation is determined by the Supporting Processor.

The logical organization of the Microprogram Memory in the run mode is shown in Figure 2.6.

The Microprogram Memory is built from 80 memory packages of the Intel 2147H type, each capable of storing 4096 bits of information. The access time is 55 ns. Data from the Microprogram Memory are loaded into the Pipeline Register (PPL) which is built from 20 4-bit registers.

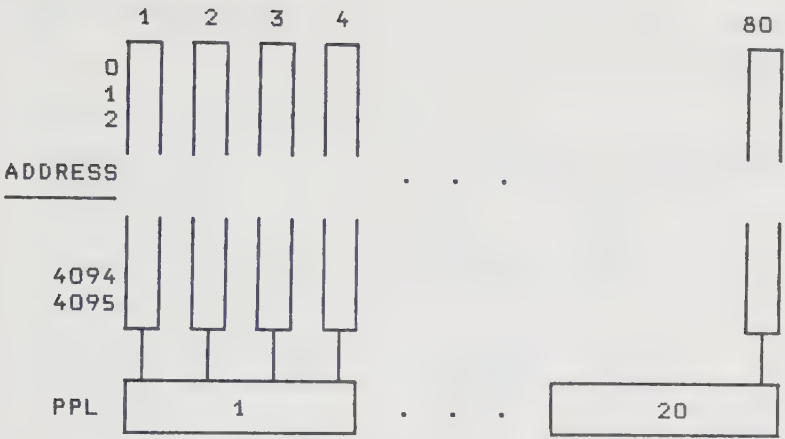


Fig. 2.6 Microprogram Memory

In the run mode, the Microprogram Memory acts as a read-only memory. Each memory package is dedicated to exactly one bit of each of the 4096 microinstruction words. When the address from the Microprogram Controller is applied, 80 bits from 80 memory packages comprising the current microinstruction word are loaded into the Pipeline Register. The part of the Pipeline Register which represents field A of the microinstruction word can be forced externally (from the front panel) to a zero state, thus outputting four zeros, which is the code of instruction JZ of the Microprogram Controller. When this microinstruction is applied, the Microprogram Controller generates the address 0 to the Microprogram Memory. This reset feature can be used during debugging of microprograms for aborting erroneous infinite loops.

In the load mode, the Microprogram Memory acts as a random access memory of 40960 bytes. It is accessed byte-wise by the Supporting Processor. During the loading of microprograms, the Microprogram Memory is disconnected from the rest of the Control Unit and from the Associative Array. The Supporting Processor applies addresses, data and control signals through a dedicated card with ports implemented by Intel 8255 components.

Mapping between the two logical organizations of the Microprogram Memory is depicted in Figure 2.7.

MICROPROGRAM WORD	0	byte 0 bit 1-8	byte 1 9-16			byte 9 73-80
	1	10 1-8	11 9-16			19 73-80
				.	.	.
	4095	40950 1-8	40951 9-16			40959 73-80

Fig. 2.7 Logical organization of Microprogram Memory

2.2.2 ADDRESS PROCESSOR

The purpose of the Address Processor is to make computations of addresses to bitslices of data in the Associative Array, and to aid the Microprogram Controller in controlling loops in microprograms.

It is very important that the Address Processor works efficiently. For example, if two fields A and B in the Associative Array are to be bit-serially added and the result stored in field C, then the following output sequence of addresses must be produced. (Ideally one in each clock cycle.)

```

address to bit 0 in A
address to bit 0 in B
address to bit 0 in C
address to bit 1 in A
address to bit 1 in B
address to bit 1 in C

```

The Address Processor, shown in Figure 2.8, consists of the following five parts:

- * An Arithmetic Unit, making all necessary computations.
- * A 12-bit address register, synchronizing the Address Processor and the Pipeline Register. This register holds the address which is currently in use, while the next one is being calculated.
- * A 32-level Address Stack memory for temporary saving of data from the Arithmetic Unit.
- * Four 12-bit Parameter Registers (P1,..., P4), receiving parameters to the microprograms from the Supporting Processor.

- * A 12-bit Loop-counter Register (LOOP CNTR), permitting loop counting to be performed simultaneously with address computation inside the Arithmetic Unit.

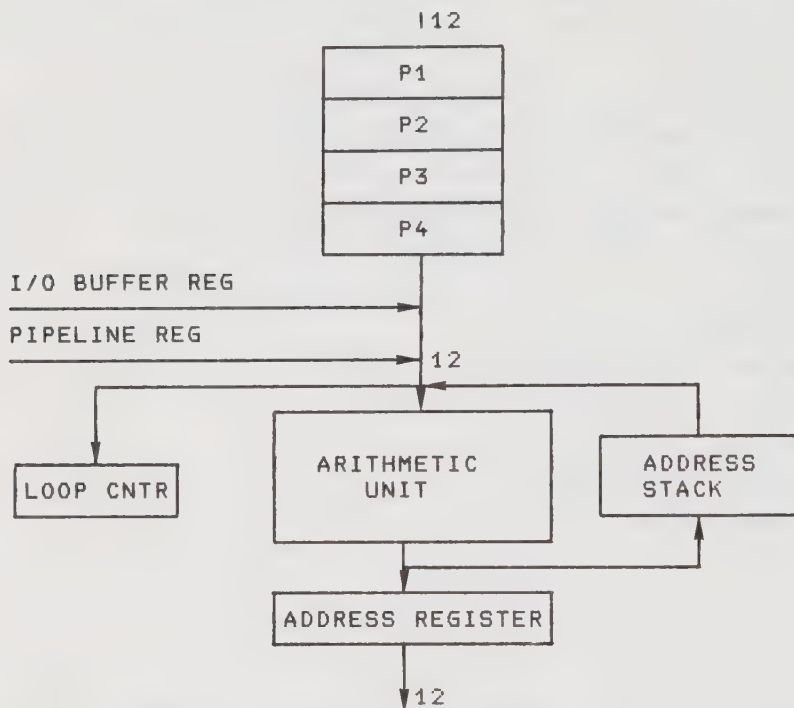


Fig. 2.8 Address Processor

The Supporting Processor sends the necessary parameters of the microprograms, e.g addresses to fields of data in the Associative Array or lengths of operands, to the four Parameter Registers. From the Parameter Registers data can be loaded into the Arithmetic Unit or to the Loop-counter Register. Loading and decrementing of the Loop-counter Register is controlled by field K of the microinstruction word. The output of the Loop-counter Register, indicating if it contains the value zero or not, is connected to the Test Selector.

The Arithmetic Unit is implemented from three 4-bit bitslice microprocessors (AM2901) connected in cascade. A simplified picture of the internal organization of the Arithmetic Unit is given in Figure 2.9.

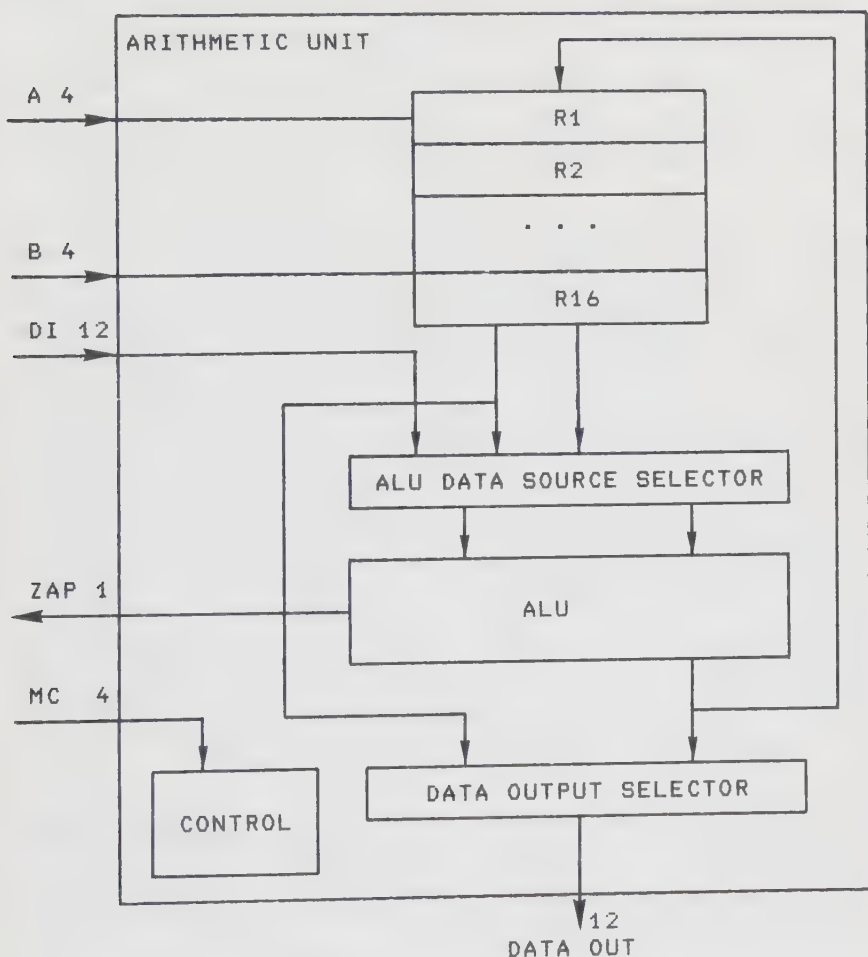


Fig. 2.9 Arithmetic Unit

The Arithmetic Unit contains 16 internal 12-bit registers ($R1, \dots, R16$) and an ALU. The source of data at the data input (DI) to the Arithmetic Unit is one of the following:

- * One of the four Parameter Registers.
- * Field H of a microinstruction word in the Pipeline Register.
- * The I/O Buffer Register.
- * The top of the Address Stack.

The source of data is selected by field N of the microinstruction word.

Because of the very specialized nature of the use of the Arithmetic Unit, the repertoire of operations that it can execute could be limited to 16. The microinstruction code is given by field E of the microinstruction word. The 4 bit microinstruction code (MC) is translated by a look-up PROM memory into 8 control signals to the bitslice processors. If the microinstruction refers to an internal register of the Arithmetic Unit, this register is pointed out by field F, the so called A-address (A), or by field G, the so called B-address (B), of the microinstruction word. The A-address is a read-only address and the B-address is both a read and write address. Any of the 16 internal registers can simultaneously be addressed by the A- and the B-address. We will refer to the register pointed out by the A-address as the A-register, and to the register pointed out by the B-address as the B-register.

The Arithmetic Unit can execute the following 16 microinstructions:

- * TBZ: The output is the value in a B-register.
- * LDBD: Data at the data input are loaded into a B-register.
This value is also output.
- * LDBA: The value in an A-register is transferred to a B-register, and is also output.
- * INCB: The value in a B-register is incremented by 1, and is also output.
- * DECB: The value in a B-register is decremented by 1, and is also output.
- * ADAD: Data at the data input are added to the value in an

A-register and loaded into a B-register. This value is also output.

- * TAEQB: The difference between values in a B-register and an A-register is output.
- * ADDBA: The values in a B-register and an A-register are added and loaded into the B-register. The sum is also output.
- * SUBBA: The difference between values in a B- and an A-register is loaded into the B-register, and is also output.
- * AINCB: The value in a B-register is incremented by 1, and the value in an A-register is output.
- * ADECB: The value in a B-register is decremented by 1, and the value in an A-register is output.
- * AADAD: The value in an A-register is added to data at the data input and loaded in a B-register. The value in the A-register is output.
- * AADAB: The value in a B-register is added to the value in an A-register and the result is loaded in the B-register. The value in the A-register is output.
- * ASUBBA: The difference between a value in a B-register and the value in an A-register is loaded into the B-register. The value in the A-register is output.
- * ALDBD: Data at the data input are loaded into a B-register, the value in an A-register is output.
- * PLADR: Data at the data input are output.

The data output from the Arithmetic Unit are loaded into a 12-bit address register. The output of this register is connected to the address inputs of all memory words in the Associative Array and also to the address inputs of the Comparand Register and the Mask Register in the Control Unit. Furthermore, data from the Arithmetic Unit can be pushed onto the Address Stack. The operation of the Address Stack (PUSH/POP) is controlled by field I of the microinstruction word. The reason for implementing the Address Stack is to extend the amount of data (addresses) available to the Arithmetic Unit during the execution of a microprogram.

Another function of the Arithmetic Unit is to support loops in the microprograms. An output of the Arithmetic Unit (ZAP) indicates if the 12-bit data value produced by the current microinstruction is equal to zero. This signal is connected to the Test Selector at the Microprogram Controller. The purpose of some of the microinstructions performed by the Arithmetic Unit, e.g. DECB or TAEQB, is not only to produce an address to the Associative Array but sometimes also to produce a signal indicating whether some loop terminating condition is satisfied or not. For example, a register in the Arithmetic Unit can be used for controlling the number of times a loop should be executed. Each time the loop is run through the register is decremented by the DECB microinstruction. When it subsequently acquires a zero value the loop will be terminated.

2.2.3 COMPARAND AND MASK REGISTERS

The purpose of the Comparand Register is to supply an argument to all Processing Elements in the Associative Array. The Mask Register is used to indicate which of two different microinstructions is to be performed on a current bitslice in the Associative Array.

The implementation of the Comparand Register and the Mask Register is shown in Figure 2.10. The size of both registers is the same as that of the memory words in the Associative Array, 4096 bits.

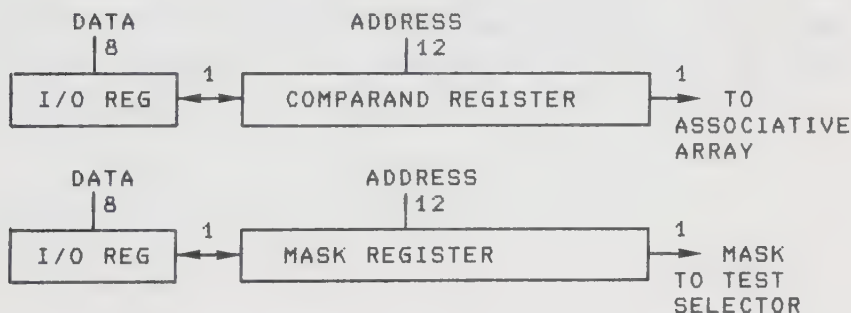


Fig. 2.10 Comparand and Mask Registers

The Comparand Register and the Mask Register are implemented in the same way. They each consist of one Intel 2147H memory package to which an 8-bit I/O Register is connected. This register, a 74LS323 shift register, is used for input and output of data.

Writing into the Comparand and the Mask Register is controlled by field L of the microinstruction word.

Data which are to be loaded into the Comparand or Mask Registers must first be written in parallel to the I/O Registers and then shifted serially into the memory packages, to addresses supplied by the Address Processor.

The output of the Comparand Register is connected to all Processing Elements in the Associative Array. During the execution of a microprogram one bit of information from the Comparand Register at the current address is transmitted to the Associative Array, where it can participate in the computation. A typical use of the Comparand Register is to hold an argument for computation in all Processing Elements, e. g. comparing it to a field in all memory words.

The output of the Mask Register is connected to the Test Selector of the Microprogram Controller. The value of the currently addressed bit can be tested by conditional microinstructions. The typical use of the Mask Register is to mask out certain bits in fields of data in the Associative Array in search operations.

2.2.4 INTERFACE

The Supporting Processor interacts with the Control Unit by reading from, or by writing into, a number of internal registers in the Control Unit. The only active part in this interaction taken by the Control Unit is when it notifies the Supporting Processor by the busy/wait signal on a control bus that it is not ready to receive new data.

The logical organization of the Interface is shown in Figure 2.11.

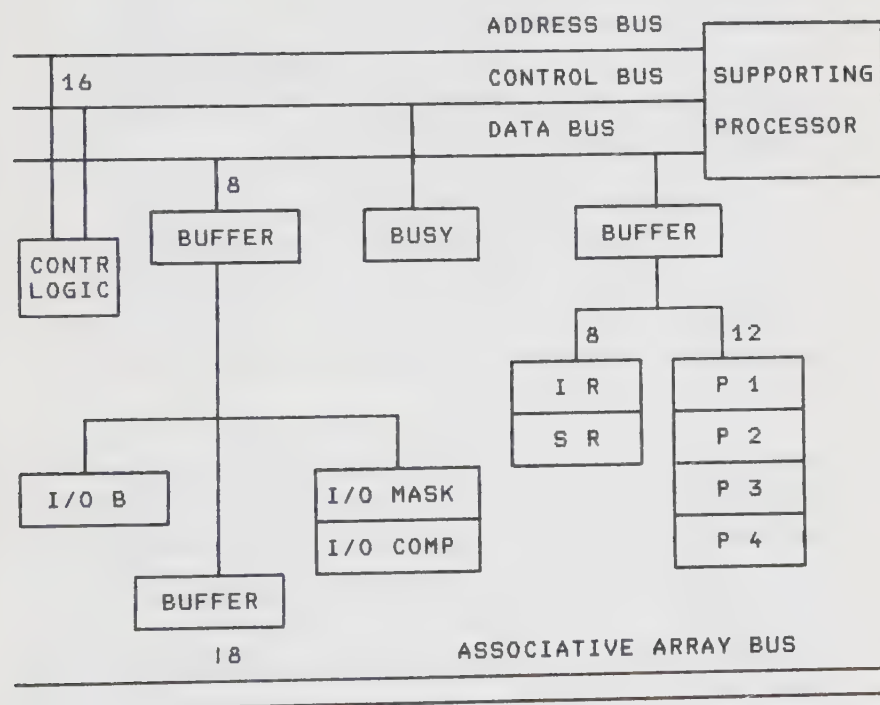


Fig. 2.11 Interface

All internal registers in the Control Unit which participate in the system communication are mapped into the address space of the Supporting Processor. Decoding circuitry (Control Logic) is implemented with fast PROM memories.

The Interface consists of the following registers:

- * An 8-bit Instruction Register (IR), which has the address FE00 (in hex code). When the Supporting Processor issues an instruction, by writing an 8-bit word into address FE00, a flip-flop in the Control Unit, called the Busy flip-flop (BUSY), is set to one. If the Busy flip-flop is set to one it prohibits loading of a new instruction into the Instruction Register and also loading of new parameters into the Parameter Registers. The Busy flip-flop is controlled by field O of the microinstruction word. Usually it is reset to zero at the beginning of the execution of an instruction. Thus, while an instruction is being executed another can be written into IR.
- * An 8-bit Status Register (SR), which has the address FE18. The Supporting Processor sees the Status Register as a read-only memory word which it can interrogate at any time. Loading data to the Status Register is controlled by field P of the microinstruction word.
- * Four Parameter Registers (P1,..., P4) which are built from pairs of 8-bit registers. The most significant nibble of the register pairs is ignored by the Arithmetic Unit. The Supporting Processor loads parameters to the microprograms by writing to addresses FE10-FE11, FE12-FE13, FE14-FE15 and FE16-FE17, respectively.
- * An 8-bit I/O Buffer Register (I/O B) which has the address FE22. The purpose of the I/O Buffer Register is to aid the intercommunication between different registers in the Control Unit and I/O Registers of processors in the Associative Array. The information in the I/O Buffer Register is produced either in the Supporting Processor (by the ordinary write

instruction) or in the Associative Array (by the LDIO instruction in field M of the microinstruction word).

- * The 8-bit I/O registers of the Comparand and Mask Registers (I/O COMP and I/O MASK) have the addresses FE20 and FE21. Data can be loaded into them either from the Supporting Processor by the write instruction, or from the I/O Buffer Register by an IOWRA microinstruction in field S of the microinstruction word.

The information in the I/O Buffer Register can be transferred to:

- * The Supporting Processor. This is used for reading out data from an associatively selected memory word in the Associative Array whose exact location is unknown.
- * The I/O Registers of the Comparand Register and the Mask Register.
- * The Loop-counter Register in the Address Processor.
- * The data input to the Microprogram Controller.
- * All I/O Registers of processors in the Associative Array.

The Status Register contains the following information:

- * The status of the Busy flip-flop.
- * Bits 63 and 64 of the microinstruction word. These bits are useful for debugging microprograms.
- * The signal NONE from the Associative Array.
- * The signal ZAP from the Address Processor.
- * The signal ZLC from the Loop-counter Register.

2.3 ASSOCIATIVE ARRAY

The Associative Array of LUCAS is dedicated to parallel and associative processing of data. It consists of an array of 128 simple processors operating under the supervision of the Control Unit. The size of the memory area is 128×4096 bits, or totally 65536 bytes. The design of the Associative Array is such that the number of processors may easily be increased with resulting improvement in performance.

2.3.1 OVERVIEW

The logical organization of the Associative Array is shown in Figure 2.12. The I/O Buffer Register (I/O B), the Comparand Register and the Mask Register are implemented in the Control Unit (their operation is explained in Section 2.2.3), but logically they are parts of the Associative Array.

CONTROL UNIT

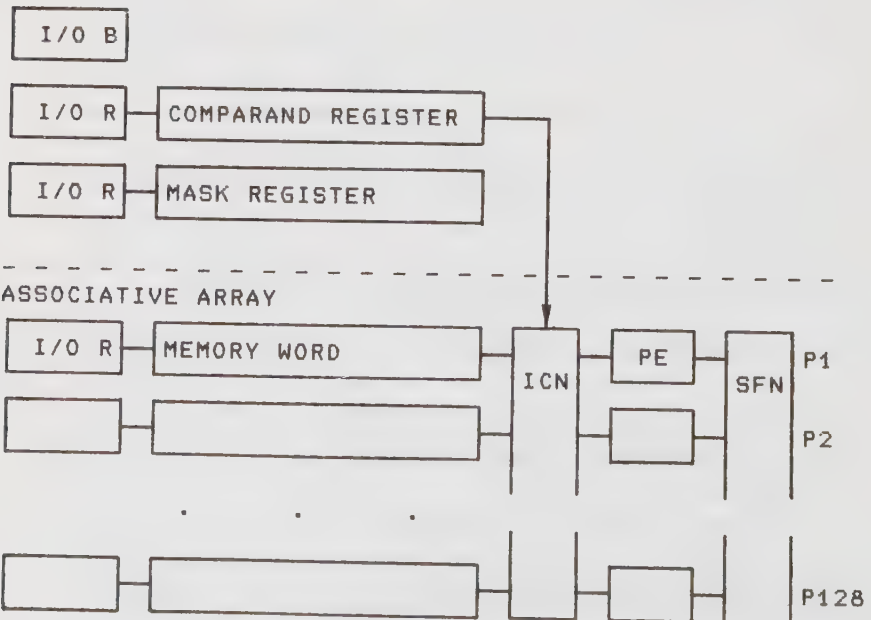


Fig. 2.12 Associative Array

Each of the 128 processors ($P_1, \dots, P_{128}$) consists of three parts:

- * A bit-serially operating Processing Element (PE) with four flip-flops for intermediate results.
- * A 4096-bit read/write memory, called a memory word.
- * An 8-bit I/O Register (I/O R), implemented with a shift register allowing parallel loading and reading.

The Processing Elements are connected to a flexible interconnection network (ICN) which provides for uniform, parallel exchange of information. The type of regular pattern assumed by the interconnection network is controlled by the microprogram. Furthermore, the Processing Elements are connected to a Select First Network (SFN) which is used for selecting one unique processor among a set of processors.

During the execution of a microprogram, all memory words receive the same addresses from the Address Processor and all Processing Elements receive the same microinstructions from the Pipeline Register. The operation of the Associative Array is of the SIMD type (Single-Instruction stream Multiple-Data stream) [Flynn72].

Though all processors receive the same instructions, data in different memory words may be manipulated differently. This can be achieved since writing to a memory word can be selectively prohibited, depending on the state of one particular flip-flop, the Tag flip-flop, in the Processing Element. The Processing Elements operate bit-serially; typically, in one clock cycle they process one bit from the memory word and one bit from their internal flip-flops, or one bit from the memory word and one bit from the Comparand Register.

The I/O Registers are mapped into the address space of the Supporting Processor at addresses FF00-FF7F. The I/O Registers are seen as ordinary memory words by the Supporting Processor, which can write bytes of data to them by ordinary write

instructions or read data by ordinary read instructions.

The I/O Registers are used for two purposes:

- * For input and output of data to the Associative Array.
The memory words can store or output only one bitslice of data at a time. The I/O Registers transform the byteslice to a sequence of 8 bitslices which can be shifted into or out of the memory words.
- * For byte oriented communication between memory words. Under the control of a microprogram, one byte in a selected I/O Register can be transferred to the I/O Buffer Register in the Control Unit.
Furthermore, the byte in the I/O Buffer Register can be written in parallel to all I/O Registers. This feature is used e.g. for transferring data from a memory word to the Comparand or Mask Registers.

The Associative Array is controlled by field R of the microprogram word. Field R is subdivided as shown in Figure 2.13.

R															
A				B			C		D	E	F	G	H		
65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80

Fig. 2.13 Microinstruction word

The subfields are dedicated to the following functions:

- * RA (bits 65-69): Instruction to the Processing Element.
There are 32 instructions, providing for arithmetic and logical bit operations.
- * RB (70-72): Control of the interconnection network.
One bit from one of eight different sources can participate in the execution of a microinstruction in the Processing Element.

- * RC (73-74): Control of writing to the memory words. One bit from one of two different sources, as described later, can be selected and written to the memory word. The write operation can be inhibited for some memory words by the T flip-flop.
- * RD (75): Control of the Select First Network.
- * RE (76): Control of a technical feature necessary for the future expansion of the Associative Array.
- * RF (77): Control of the I/O Registers. The I/O Registers can be shifted, their output can be selectively disabled etc.
- * RG (78): Control of writing to the memory words.
- * RH (79-80): Control of the I/O Registers.

Fields RC and RG, and also fields RF and RH, are always used together, the reason why they are separate is just technical.

2.3.2 PROCESSOR IN THE ASSOCIATIVE ARRAY

The organization of one processor of the Associative Array is shown in Figure 2.14.

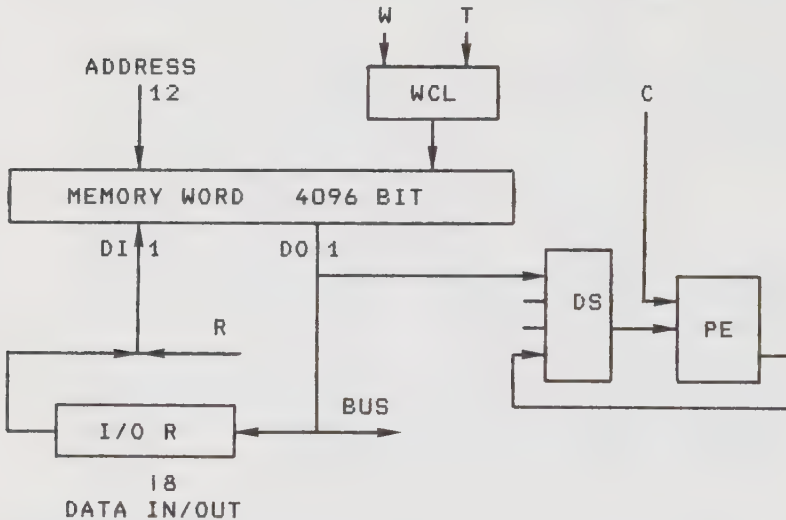


Fig. 2.14 Processor

Each memory word is implemented by an Intel 2147H 4096x1-bit memory package. The bit of data to be written into the the memory word (DI), at a 12-bit address supplied by the Address Processor, may originate from two sources:

- * The I/O Register (I/O R). This source is used for loading data from the Supporting Processor or the I/O Buffer Register.
- * A dedicated flip-flop (R) in the Processing Element (PE), called the R flip-flop. This source is usually used during computation.

Writing to the memory word is controlled by a write control logic (WCL). The inputs to the write control logic are:

- * Fields RC and RG of the microprogram word. (W).

- * The Tag flip-flop (T).

There are two types of write operations:

- * WRITE ALL The bit at the input is written to the memory word independently of the state of the Tag. This means that when this microoperation is executed a bitslice of data is written to all memory words of the Associative Array.
- * WRITE The bit at the input is written only if the Tag is set to one.

The destination of a bit of data read from the memory word (DO) may be:

- * The I/O Register. This destination is, for example, used when data from the Associative Array are to be transferred to the Supporting Processor or the Comparand Register.
- * An input to a Data Selector (DS), which is a part of the interconnection network. The Data Selector supplies one argument to the Processing Element.
- * The 128-bit wide bus (BUS). This bus is a part of the interconnection network.

Data can be loaded to the I/O register either in parallel, from the Supporting Processor or the I/O Buffer Register, or serially, by shifting bits to it from the memory word.

Figure 2.15 shows the organization of the Processing Element.

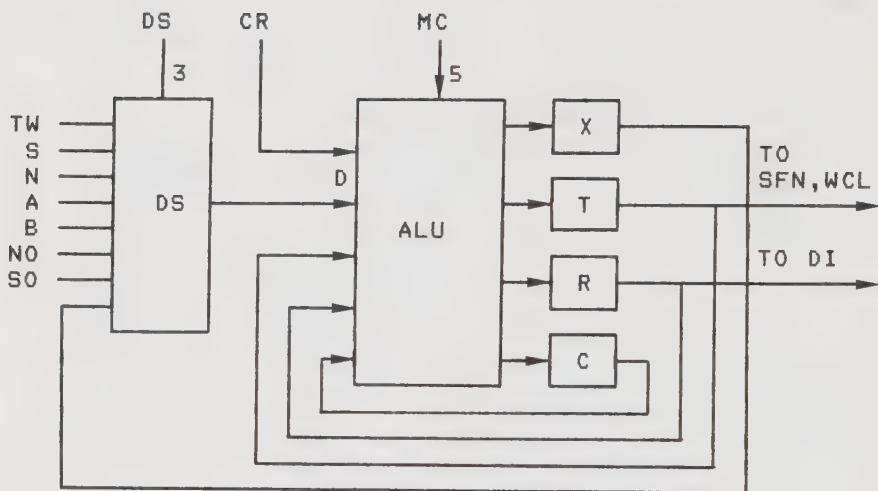


Fig. 2.15 Processing Element

The central part of the Processing Element is the ALU. The ALU is implemented by a bipolar programmable read-only memory (PROM) of type Harris 7643 with an access time of 60 ns. The size of the PROM is 1024 words of 4 bits.

The 5-bit instruction code (MC) and five bits of data are applied to the ten address inputs of the PROM. The 4-bit data word which makes the output is interpreted as the result of the microinstruction. The result is loaded into four flip-flops T, R, C, and X.

Since the ALU is implemented by a PROM which can easily be replaced, the instruction repertoire of the Processing Element can be changed to suit different applications.

The address inputs of the PROM are divided into four groups:

- * Five bits with the instruction code (MC). There are 32 instructions supplied by field RA of the microinstruction word.

- * The bit from the Comparand Register (CR) of the current bit address issued by the Address Processor.
- * The bit (D) from the output of the Data Selector (DS). The Data Selector is implemented by a multiplexer. It is a part of the interconnection network. The source of the data it outputs is determined by field RB of the microinstruction word in the Pipeline Register.
- * Three bits with intermediate data from the T, R, and C flip-flops.

The flip-flops of the Processing Element are assigned to different tasks:

The T flip-flop is connected to the write control logic (WCL). Its content, if zero, can inhibit writing to the corresponding memory word. Furthermore, the Tag is connected to the Select First Network (SFN). The Select First Network is activated by field RD of the microprogram word in the Pipeline Register. Its purpose is to select one processor among all processors with their Tags set to one, according to an arbitrary but fixed linear order. This feature is used e.g. if some data in a number of words are to be processed sequentially. The result of the selection microoperation is that all the Tags, except for the one of the selected processor, are reset to zero. The Select First Network is implemented by a chain of OR and AND gates. It is fed by outputs of the Tags and produces the reset signals to the Tags. The Select First Network also includes a chain of gates which produce the signal NONE. This signal is connected to the Test Selector of the Control Unit. It indicates whether or not there are any Tags in the Associative Array with the value one. This information may be used to influence a flow of microprograms.

The R flip-flop is used as a result flip-flop. Data from the R flip-flop may be written into the memory word.

The C flip-flop is typically used in bit-serial arithmetic operations

for holding the carry bit.

The X flip-flop is connected to the multiplexer of the interconnection network. Its typical use is for saving the value of the Tag.

The interconnection network provides means for bit-oriented parallel communication between processors in the Associative Array. The outputs from all memory words, suitably buffered, are collected on a 128-bit wide bus, and from this bus they are connected via a strapping area to the Data Selectors in the processors according to some useful interconnection patterns. Two of the eight inputs to the Data Selector are connected to fixed sources, the rest can be connected to the bus in many ways, to suit different applications.

Typically, the following eight signals are connected to the inputs of the Data Selector. In our particular application, only the first two are used.

- * Data from the memory word of the same processor (TW), a fixed input.
- * Data from the X flip-flop of the same processor. This input is fixed.
- * Data from the memory word of another processor according to the perfect shuffle interconnection pattern (S)[Stone71].
- * Data from the shuffle input of the Data Selector of one of the neighbouring processors.
- * Data from the memory word of the processor above, according to a linear order of processors (A). The order is defined by the Select First Network.
- * Data from the memory word of the processor below (B).

- * Data from the memory word of the processor to the north, according to a quadrant grid interconnection pattern (NO).
- * Data from the memory word of the processor to the south (SO).

To illustrate the potential of the processors in the Associative Array for innumerable bit-serial algorithms, we give in Table 2.1 a list of instructions implemented in the ALU. The table gives symbolic names of instructions and shows the results which they produce, namely the data to be loaded into the T, R, C, and X flip-flops. In the table, D is the bit from the Data Selector, its value originating usually in the memory word in the same processor, CR is the bit from the Comparand Register, CY is the carry produced by an addition microinstruction, OV is the overflow, BO is the borrow in a subtraction microoperation.

MNEMO	next T	next R	next C	next X	comment
NOP	T	R	C	X	no operation
COT	T/	R	C	T	complement T
SCA	T	R	1	D	set carry to 1
CCA	T	R	0	D	clear carry
SETT	1	R	C	T	set Tags
CORA	T	R/	C	D	complement R
CRA	T	0	C	D	clear R
LTRIT	T AND R/	R	C	D	if T=1, load T from R, complemented
LTRT	T AND R	R	C	D	T and R
LXMA	T	R	C	D	load X from D
LRMA	T	D	C	R	load R from D
LTMA	D	R	C	T	load T from D
LTMT	T AND D	R	C	T	if T=1, load T from D
LTMIT	T AND D/	R	C	T	if T=1, load T from D, complemented
LTRA	R	R	C	D	load T from R
LCRA	T	R	R	D	load C from R
LRTA	T	T	C	D	load R from T
LRCA	T	C	C	D	load R from C
XRT	R	T	C	D	exchange T and R
CMCT	T AND CO XOR D	R	C	T	if T=1, compare Comparand with D
CRCT	T AND CO XOR R	R	C	T	if T=1, compare Comparand with R
CRMT	T AND R XOR D	R	C	T	if T=1, compare R with D
ANDRMA	T	R AND D	C	R	R and D
XORRMA	T	R XOR D	C	R	compare R with D
XORRTA	T	T AND R XOR T	C	R	if T=1, compare R with T
ORRMA	T	R OR D	C	R	R or D
ADMA	T	R+D+C	CY	OV	addition of R and D
SUMA	T	R-D+C	BO	OV	subtraction between R and D
ACMA	T	CO+D+C	CY	OV	addition of Comparand and D
SCMA	T	D-CO+C	BO	OV	subtraction between D and Comparand
ADMIA	T	R+D/+C	CY	OV	addition of R and D/
ASMT	T	R+D+C R-D+C	CY BO	OV OV	if T=1 addition if T=0 subtraction

Tab 2.1 ALU instructions

2.4 SYSTEM OPERATION

This section gives a brief description of the operation of LUCAS. For further details see [Fernström83], [Svensson83].

2.4.1 PROGRAM EXECUTION

The system, consisting of the Supporting Processor on one side and the Control Unit and Associative Array on the other, operates in simple master-slave fashion. The Associative Array acts as an attached resource to the Supporting Processor.

An application program on LUCAS consists of two parts. One part is executed on the Supporting Processor, the other part is executed in the Control Unit on data in the Associative Array. Both parts can be executed concurrently, but sometimes the Supporting Processor must wait for the Control Unit to finish some instruction, or the Associative Array stays idle when not needed.

The part of the program which is executed by the Supporting Processor may be written in Z-80 assembly language or in UCSD Pascal. The part executed in the the Associative Array consists of microprograms. LUCAS has no fixed instruction set, the Microprogram Memory is writable and the microprograms suiting given applications are loaded from the secondary storage at system start up.

The Supporting Processor issues an instruction by writing it into the Instruction Register of the Control Unit. The instruction is fetched by the Microprogram Controller and executed by a microprogram. If the instruction needs any parameters, they must be sent to the Control Unit prior to the instruction. For example, the operation ADD FIELDS adding two b-bit fields A and B in the Associative Array and storing the result in a field C has four parameters: address to A, address to B, address to C, and the field size, b. Before this instruction can be issued, the

Supporting Processor must write the parameters into the Parameter Registers. At the beginning of the execution of ADD FIELDS, the parameters are loaded into registers of the Arithmetic Unit of the Address Processor to be used during the computation.

Very often, there are alternative ways to optimally assign hardware resources of the Control Unit for efficient implementation of an instruction. For example, the field size in the ADD FIELDS instruction, determining how many times a loop must be executed when adding two bitslices, could be loaded into registers in the Arithmetic Unit or into the Loop-counter Register.

The hardware provides only four Parameter Registers and consequently only a maximum of four parameters may be sent to the Control Unit prior to an instruction. But many complex operations must have more than just four parameters and this hardware restriction is solved by the following method. Complex operations are implemented without explicit reading of parameters from the Parameter Registers to the Address Processor. A microprogram of such a complex operation assumes that parameters are already loaded by a series of simple instructions with parameters such as 'Copy the Parameter Register x to the Address Processor Register y'.

When a microprogram for one instruction is executed, the Control Unit immediately starts executing the microprogram Instruction fetch. This consists of two microinstructions: first, a conditional jump to itself depending on the state of the Busy flip-flop (this implements waiting for arrival of a new instruction from the Supporting Processor), and, second, a JMAP instruction to the address given by the instruction code in the Instruction Register.

The Instruction Register acts as a one stage instruction pipeline. When the Control Unit starts execution of one instruction which subsequently resets the Busy flip-flop, the next instruction can be loaded into the Instruction Register by the Supporting Processor. If the Supporting Processor tries to load a third

instruction to the Instruction Register while the first is still running, it is forced to a wait state by the Busy flip-flop. It resumes operation only after the instruction which is currently in the Instruction Register is accepted and resets the Busy flip-flop again.

In the case when it is not desirable that the Supporting Processor stops execution while waiting for the Control Unit to finish the execution of some instruction, a polling method can be used. The Supporting Processor can periodically interrogate the Status Register of the Control Unit, which contains the state of the Busy flip-flop, by its ordinary read instruction. If the Busy flip-flop contains a zero value, then the Supporting Processor writes the next instruction into the Instruction Register. If the value is one the Supporting Processor carries out other tasks.

There is also another mode of cooperation between the Supporting processor and the Control Unit in feeding instructions to the Instruction Register. The Busy flip-flop may control the interrupt signal to the Supporting Processor and request the new instruction when the present one is finished. This mode of operation is used when the execution of a microprogram by the Control Unit takes much longer time than the execution of the program task in the Supporting Processor supplying new instructions.

2.4.2 DATA I/O

The Associative Array is used optimally when it is operating in parallel on large volumes of data. The data are brought from a secondary memory and, after some processing in the Associative Array, returned and saved in the secondary memory. A consequence of this type of operation is that the I/O traffic is periodically very heavy. In order that it will not become the

bottleneck of the system performance, the input and output of data must be implemented efficiently. At the time of writing this thesis, a fast dedicated I/O processor is designed but not yet implemented. Currently, the following two, less satisfactory, methods are used.

The first method is used for block transfer to memory words which are explicitly addressed by the Supporting Processor. Data are first loaded into the primary memory of the Supporting Processor from the floppy-disk. Then, by simply moving bytes from one set of addresses in the primary memory to another set of addresses, the I/O Registers which are memory mapped are filled with a byteslice of data. Next, the Supporting Processor writes into one of the Parameter Registers of the Control Unit the 12-bit destination address of the data. Finally, it issues the instruction STORE I/O REGISTERS which transfers the content of the I/O Registers to all memory words. In this way, one byteslice is loaded and the procedure is repeated. This method does not use any associativity, data are simply transferred from one area to another.

The second method is used for associative transfer of data to memory words which are determined by the Tags and with their exact identity unknown to the Supporting Processor. A byte is moved from one address in the primary memory to another which is the address of the I/O Buffer Register. Then the byte in the I/O Buffer Register is transferred in one clock cycle to the I/O Registers. All 128 I/O Registers hold the same byte, and by the instruction STORE I/O REGISTERS[*T*] this byte is shifted to all memory words with the Tag set to one.

Methods for outputting data from the Associative Array are similar.

The Associative Array of 128 processors resides in the top 19" rack. It consists of 16 220x230 mm double layered boards. Each board houses 8 processors implemented from a total of 70 IC packages and consumes approximately 3 amperes. This rack also holds two other boards, B containing a bus termination circuitry, and A which is used for interconnection of the backplane bus of the rack with the Control Unit.

Below the Associative Array, the rack with the Control Unit resides. It contains four boards: A, B, C, and D. A is used for interconnection of the Control Unit with the Associative Array and with the Supporting Processor. B contains the wirewrapped Control Unit. C is a board with the circuitry necessary for the loading of microprograms into the Microprogram Memory. Finally, D houses the Microprogram Memory and the Pipeline Register. The Microprogram Memory consumes 18 amperes.

The next rack contains the Supporting Processor. A is the board dedicated to internal communication between the racks. Boards B, C, D, E, and F are standard Prolog cards comprising the Z-80 microprocessor system.

Between the Associative Array, the Control Unit, and the Supporting Processor racks there are cooling fans which are used for dissipating the heat generated by LUCAS.

The two bottom racks are occupied by a floppy-disk system, and power supplies.

Chapter 3.

DATABASE SYSTEMS

In this chapter we will give a brief orientation in the area of database technology by way of informal definitions of some of the more important concepts which will be illustrated by simple examples.

The purpose of the presentation is to provide a general framework for the material in subsequent chapters.

3.1 BASIC CONCEPTS

In the literature about information processing many similar definitions of the concept of a database can be found. We give the following two.

* A database is a collection of interrelated data stored together with controlled redundancy to serve one or more applications in an optimal fashion. The data are stored so that they are independent of programs which use the data. [Martin76]

* A database is a collection of stored operational data used by the application systems of some particular enterprise. [Date81]

Any organization or enterprise which must record and maintain some large amount of information for its successful operation ought to organize this information in a database. Typical users of databases are banks, government agencies, hospitals, schools, companies etc.

As an example, let us look at the information processing in a large manufacturing company. Company management at different levels needs access to a large amount of information of different types in order to be able to make rational decisions governing activities of the enterprise. At the highest level of the managing hierarchy of the company, long range strategical plans about future products and new markets are made. The middle level management is concerned with optimal allocation of resources of the company, and at the lowest level, at the factory floor, detailed day to day planning is conducted, e.g. of the use of some machine or robot. At the same time a number of other data processing functions within the company are going on. The payrolls are computed, production statistics are compiled, banking transactions are initiated etc. All those activities need access to a large reservoir of information which includes data about employees (name, address, age, position, skills etc), data about suppliers (address, status, type of products supplied, delivery times etc), data about products, data about production facilities, data about customers, data about current projects, data about markets etc - this list can be made very long.

What makes this reservoir of data into a database instead of simply being a collection of distinct files are two features: data should be integrated and shared. This means that in the database all information is in some way logically unified. Different users may access the same subsets of the database, and the same items of data, while having different views of the data and using the data for different purposes.

There are basically four different operations on the data in the database. Data are created (e.g. when information about a new customer is added), data are deleted (e.g. when a contract with a

supplier is expired and will not be renewed), data are modified (e.g. when an employee has been promoted with a subsequent change of his salary), and finally, data are retrieved (e.g. when a sales manager wants to know addresses of all the customers in England).

In general, the database has three groups of users with different objectives.

- * The first group is the database administrator, DBA. This is the person in charge of the database. He decides how the database is organized, how data are represented and stored, how data are accessed, who is authorized to do what with the information in the database, how a system crash is handled etc. In a large company the DBA is a group of persons and the head of this group, because of his crucial importance for the company, is usually a member on the highest management level.
- * The second group of users are the application programmers. They write programs (e.g. in COBOL or FORTRAN) for processing information in the database (e.g. computing the payrolls).
- * The third, very heterogenous group of users are the end-users. It is for the end-user that the database is created. He is the proper user of the database. A very important feature of the end-user is his ignorance. The end-user does not have to know how a computer works, how to program it or even where the computer is located. Also, the fine details of the organization of the database are hidden from him. The end-user either invokes application programs written by the application programmer for performing some data processing services (e.g. computing sales statistics) or he interacts with the contents of the database from a terminal with the help of a special, usually very easy to learn, language called a query language.

The agent responsible for managing the information in the database is a software and hardware system called the database management system, DBMS. All users access the database using services of the DBMS. Figure 3.1 gives a simple illustration of the use of the DBMS.

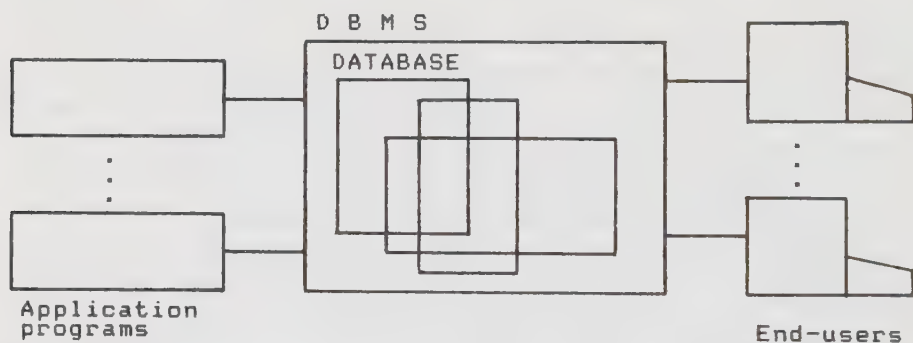


Fig. 3.1 Users of DBMS

The database together with its associated DBMS, the host computer and the users form a highly complex dynamical system. The architecture of this system can be analyzed from different points of view. A particularly important way of describing it is depicted in Figure 3.2 where the system architecture is decomposed into three levels.

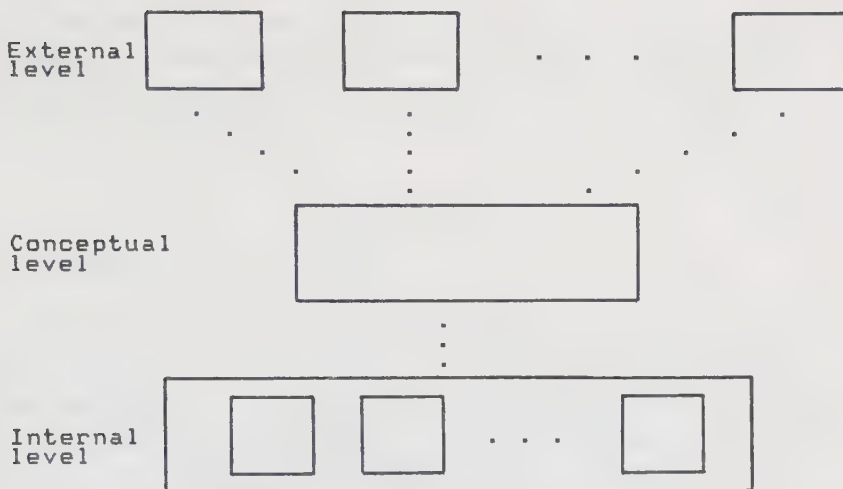


Fig. 3.2 System Architecture

- * The internal level is the level concerned with the way data are represented on the storage medium. A mapping between the internal level and actual physical occurrence of data in storage is of the greatest importance for the performance (speed, utilisation) of the DBMS.
- * The highest level of the architecture is the external level. It is the level of the end-users with their different external views. The external view can be seen as the user's window into the database. Furthermore, it is also his understanding and his tools to manipulate the contents of this window.
- * Between the external level of users and the internal level of storage structures there is a conceptual level. It is the global level of the information of the entire database. At this level, data are described in a more abstract form than at the internal level and more truthfully than at the external level. In its simplest form it is the sum of all users' views together with some

additional information or procedures. The conceptual level is the universe of discourse of a DBA. The DBA also defines the mappings between the conceptual and the other two levels.

3.2 DATA MODELS

Data in a database represent a description of some aspects of the real world. This description is usually highly structured. The type of structure which is used is called the data model. The three most important and most widely used data models in database systems are the hierarchical data model, the network data model and the relational data model.

In this section we give simple examples of all three data models. The model most relevant to our research is the relational data model. A more detailed presentation of it is given in the next section.

We illustrate the data models with the help of a simple database concerning students who are enrolled in a VLSI design course, and components that the students are to design. The information in the database reflects the following situation:

- * Four students, Petra, Jon, Nicolas and Filip are currently enrolled in the course.
- * During the course they must design three types of components, an inverter, a nand gate and a multiplexer.
- * Petra and Nicolas registered at the University in 1978, Jon and Filip in 1976.
- * Difficulty ratings are: for an inverter design 1, for a nand gate design 2, and for a multiplexer design 4.

* Petra has finished a design of the inverter and is now working on the nand gate, Jon is working on the nand gate, Nicolas has finished the inverter and Filip is working on the inverter.

3.2.1. HIERARCHICAL DATA MODEL

In a hierarchical data model, the data are structured as a forest of trees. Nodes in a tree stand for entities and a parent-child relationship between the nodes mirrors some associations between the entities.

Figure 3.3 depicts one possible representation of the sample database in the hierarchical model.

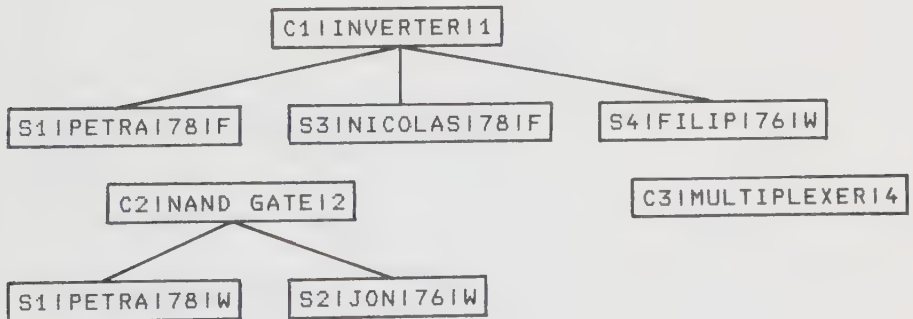


Fig. 3.3 Hierarchical data model

Because a hierarchical organisation is a very common phenomenon in the real world, the hierarchical data model for describing features of the real world is very natural and is widely used.

But this model is afflicted by many undesirable properties, some of which can already be demonstrated on our simple example. E.g. where can we add to the database information about student Petter who has not started working on any project yet, or what happens to the information about Nicolas and Filip if we decide that the design of the inverter is too trivial to be a design project? Those problems are caused by the fact that there is an asymmetry between superiors

superiors. By removing the information about superiors we lose the information about dependents.

3.2.2. NETWORK DATA MODEL

A network data model is more general than a hierarchical data model. The data are structured as a directed graph where nodes represent entities and arcs are associations between the entities.

Figure 3.4. depicts one possible representation of the sample database in the network model.

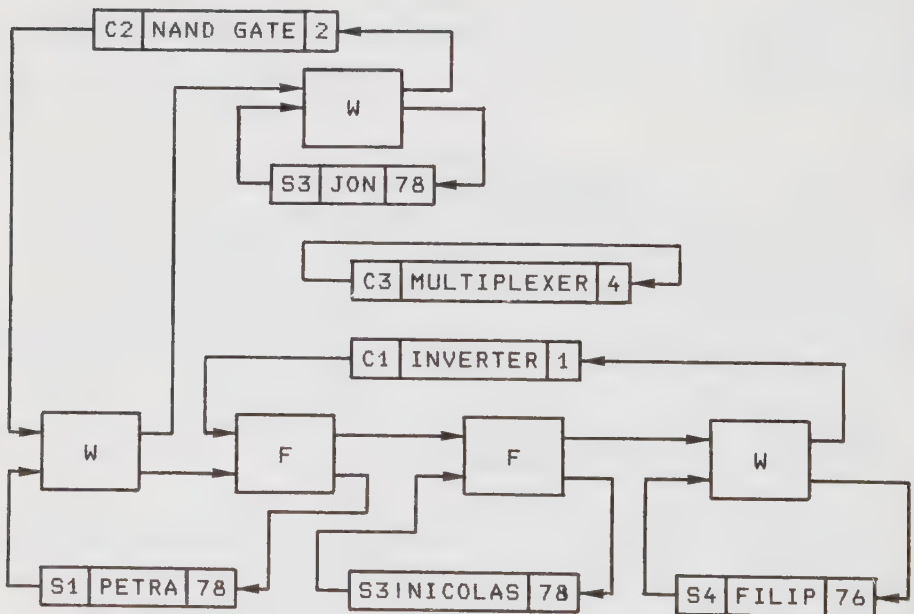


Fig. 3.4 Network data model

The problems of asymmetry which we encountered in the hierarchical model do not exist in the network model. The greatest disadvantage of the network model is its complexity.

3.2.3. RELATIONAL DATA MODEL

In a relational data model the data are organized as a collection of tables, called relations. Rows of the relation correspond to entities, columns to properties of the entities. The association between the entities in different relations is mediated by values in columns drawn from a common domain.

Figure 3.5. depicts one possible representation of the sample database in the relational data model.

S:	S#	NAME	YEAR
	S1	PETRA	78
	S2	JON	76
	S3	NICOLAS	78
	S4	FILIP	76

C:	C#	ITYPE	DIFFICULTY
	C1	INVERTER	1
	C2	INAND GATE	2
	C3	MULTIPLEXER	4

SC:	S#	C#	STATUS
	S1	C1	F
	S1	C2	W
	S3	C1	F
	S4	C1	W
	S2	C2	W

Fig. 3.5 Relational data model

A typical feature of the relational data model is that all the information that the database captures is represented in only one way, in the form of relations. This conceptual simplicity (no need for pointers) is the main advantage of this model. All the information processing consists of manipulation of the relations. Information is added by adding new rows or columns to the relations or even by creating new relations. Information is deleted by deleting rows or columns from the relations.

3.2.4 COMPARISON OF THE THREE MODELS

Hierarchical and network data models are prevailing among commercial database systems. But this situation is gradually changing. The relational data model is becoming more and more popular and will probably be the most common approach to database organisation in the future. [Codd82]

The two most important criteria for judging the merits of different data models are ease of use and efficiency of implementation. [Ullman1980]

- * According to the first criterion - ease of use, the relational data model is clearly superior to the other two. There is only one simple concept, a relation, that an application programmer and an end-user must understand.
- * According to the second criterion - the efficiency of implementation, measured by the time needed by a computer to execute demands of users and by the amount of storage required by the database - the comparison between data models is not so simple. For very large databases where the type of operations on data can be anticipated in advance, hierarchical and network data models can be implemented more efficiently than a relational data model. For smaller databases, and they are in a majority, or for evolving databases where future use is not well-defined the efficiency of the relational model can be as high as for the other two data models.

Last but not least, a database organized according to a relational data model gives more opportunities for implementation with the help of special hardware (e.g. associative array) in the database computer than when it is organized according to the other two models.

3.3 RELATIONAL DATABASE

The development of a relational data model started in the mid-1960s when a number of researchers described a method for structuring data as a set of tables, each table representing entities of the same type. The most important single contribution was the pioneering work of E. F. Codd [Codd70], who noted that the data tables can be viewed as relations (the well known mathematical objects of set theory) and thus established a firm theoretical base for further research and development.

The motivation for Codd's [Codd82] research, resulting in a relational data model, was to meet three objectives:

- * The data independence objective, of providing a sharp and clear boundary between the logical and physical aspects of database management.
- * The communicability objective, providing different kinds of users with common understanding of the database through structural simplicity of data, and making it possible for them to communicate with each other about the database.
- * The set-processing objective, of introducing powerful high level language concepts to enable users to express operations on large amounts of data at a time without using iteration or recursion.

3.3.1 RELATIONS

The precise definition of a set theoretical relation is the following:

Given a collection of sets $D_1, D_2, \dots, D_n$ a relation R is a subset of a Cartesian product $D_1 \times D_2 \times \dots \times D_n$. (The Cartesian product $D_1 \times D_2 \times \dots \times D_n$

is the set of all ordered n -tuples $\langle d_1, d_2, \dots, d_n \rangle$ where each d_i is selected from the set D_i . The sets $D_1, D_2, \dots, D_n$ are called domains of R and they do not necessarily have to be distinct. N -tuples of R are called tuples.

In a database, the relation R is a time varying finite subset of a Cartesian product of its domains.

There is always need to identify uniquely and uniformly every tuple within a given relation. This purpose is served by one attribute, or a combination of attributes, with a unique value in each tuple. This attribute or this combination of attributes is called a key. A relation can have several keys and it may be the case that the only key it has is a combination of all its attributes.

A relation is a set and this implies that there is no ordering on tuples of which it consists. This property is very important and is one of the reasons for success of the relational data model.

In the popular 'table' interpretation of a relation, a relation is a table where tuples are rows and where each component of the tuples corresponds to a column. The columns are called attributes. Typically, the rows represent entities of the real world and values of attributes are properties of these entities. The number of tuples in the relation is called the cardinality of the relation and the number of attributes is called the degree of the relation.

Relations in a database must satisfy a number of conditions to be of practical use. A process of imposing restrictions on relations, eliminating undesirable properties, is called a normalization. A relation is said to be in the first normal form if each value in each tuple is atomic, which means that sets are not allowed as elements of domains.

A relational database is defined as a collection of relations in the first normal form.

3.3.2. DATA MANIPULATION LANGUAGES

An end-user of a database needs a language for expressing his requirements. The most significant aspect of this language is its capacity for expressing queries to the database. In general, a query is a function applied to a collection of relations and its purpose is to retrieve information from the database.

The end-user's language is often called a query language.

A query language has usually two sublanguages, a data definition language DDL, for defining type and format of data and a data manipulation language DML, supporting manipulation of the data.

The other three operations on the database, namely creation, deletion and modification of data are in most cases expressed with the help of the query language.

According to the way in which they express queries, query languages may be divided into two large groups. The first group is relational calculus languages where the result of the query is specified by giving a predicate which must be satisfied by the tuples of the result relation. The relational calculus languages define the result of the query in terms of the relations of the database but do not specify how to obtain it, giving the DBMS freedom to choose the best strategy. The second group is algebraic languages where sentences of the language define a sequence of operations on given relations. A strategy to obtain the result of the query is explicitly described.

It can be shown that most query languages in both groups are equivalent in their expressive power. This property is called relational completeness: if a query can be stated in one language then it is possible to express it in the others as well.

3.3.3 RELATIONAL CALCULUS

Relational calculus languages are based on the well-known language of first order predicate logic. The formalism may be different in different languages but the general form of expression is $\{x : W(x)\}$, where x is the only free variable in the well-formed formula W . This expression is interpreted as stating that the result of the query $W(x)$ is the set of tuples x for which $W(x)$ is true.

Depending on the range of the variables in the well-formed formulas the relational calculus languages can be classified as tuple relational calculus languages or domain relational calculus languages. In the first case the range of the variables is tuples, and in the second case it is domains of attributes of the relations.

3.3.4 RELATIONAL ALGEBRA

Relational algebra is a collection of operations on relations. Each operation takes one or two relations as its operands and produces another relation as a result. The operations may in principle be nested to any depth in an algebraic expression.

In this section we present algebraic operations on relations. The presentation will be informal and it is based on simple examples. The rigorous definitions of most of the operations can be found elsewhere e.g. in Principles of Database Systems by J. D. Ullman. [Ullman80]

It can be proved that any language which includes these operations is relationally complete.

We will also introduce a variant of the Relational algebra notation which we call the Algebraic Assignment language. This notation is very convenient and will be used in Chapter 6, as a tool for expressing a sequence of computations in the Associative Array during an evaluation of a query to a database.

SELECTION

A Selection operation yields a result which is a horizontal subset of tuples of a given relation. Each tuple of the result must satisfy a specified predicate. Typically, the predicate states that values of an attribute must be equal to some datum.

In Figure 3.6 we can see two relations. R is the source relation with names of some persons and places where they live, and T is the result of the Selection on R where the attribute CITY has the value LONDON. In the Algebraic Assignment language notation we write

$T := R \text{ WHERE CITY} = \text{'LONDON'}$

R:

NAME	CITY
SMITH	LONDON
JONES	PARIS
BLAKE	PARIS
CLARK	LONDON
ADAMS	ATHENS

T:

NAME	CITY
SMITH	LONDON
CLARK	LONDON

Fig. 3.6 Selection

The result T is the relation with the information about persons living in London.

INTERSECTION

An Intersection operation yields a result which is a set of tuples belonging to two given relations. The source relations must be of the same type, which means that they have the same degree and that values in their corresponding attributes are drawn from the same domains. However, the names of the attributes may be different.

In Figure 3.7 we can see three relations. R1 and R2 are source relations and T is the result of the Intersection between R1 and R2.

In the Algebraic

Assignment language notation we write

$T := R1 \text{ INTERSECTBY } R2$

R1:

NAME	BORN
SMITH	LONDON
JONES	LONDON
BLAKE	BERLIN
ADAMS	PARIS
CLARK	PARIS

R2:

NAME	LIVE
SMITH	LONDON
JONES	PARIS
CLARK	LONDON
BLAKE	PARIS
ADAMS	PARIS

T:

NAME	CITY
SMITH	LONDON
ADAMS	PARIS

Fig. 3.7 Intersection

If we assume that R1 contains the information about places of birth of some persons and R2 information about where they currently live, then the Intersection between R1 and R2 gives T with the information about persons living in the city where they are born.

UNION

A Union of two relations of the same type is a relation consisting of tuples belonging to at least one of the source relations.

In Figure 3.8 we can see three relations. T is the result of the Union between R1 and R2. In the Algebraic Assignment language notation we write

$T := R1 \text{ UNION } R2$

R1		R2		T	
NAME	CITY	NAME	CITY	NAME	CITY
SMITH	LONDON	JONES	PARIS	SMITH	LONDON
CLARK	LONDON	BLAKE	PARIS	JONES	PARIS
BLAKE	PARIS	ADAMS	ATHENS	BLAKE	PARIS
				CLARK	LONDON
				ADAMS	ATHENS

Fig. 3.8 Union

If R1 represents a group of persons having some shared property and R2 persons with another property, then T represents persons having at least one of those properties. (The property can be as simple as membership in a group.)

DIFFERENCE

A Difference between two relations of the same type is a relation consisting of tuples belonging to the first but not to the second relation.

In Figure 3.9 we can see three relations. T is the result of the Difference between R1 and R2. In the Algebraic Assignment language notation we write

$T := R1 \text{ MINUS } R2$

R1

NAME	CITY
SMITH	LONDON
CLARK	LONDON
BLAKE	PARIS

R2

NAME	CITY
JONES	PARIS
BLAKE	PARIS
ADAMS	ATHENS

T

NAME	CITY
SMITH	LONDON
CLARK	LONDON

Fig. 3.9 Difference

T represents the information about persons from R1 not having the property represented in R2.

PRODUCT

A Product of two relations is a relation consisting of tuples being a concatenation of all combinations of tuples from the first and second relation.

In Figure 3.10 we can see three relations. T is the result of the Product of R1 and R2. In the Algebraic Assignment language notation we write

$T := R1 \text{ TIMES } R2$

R1		R2		T	
NAME		CITY		NAME	CITY
SMITH		LONDON		SMITH	LONDON
CLARK		PARIS		SMITH	PARIS
BLAKE				CLARK	LONDON
				CLARK	PARIS
				BLAKE	LONDON
				BLAKE	PARIS

Fig. 3.10 Product

PROJECTION

A Projection of a relation equals elimination (or rearrangement) of some of its attributes and removal of any redundant tuples. The result of the Projection is a vertical subset of the source relation.

In Figure 3.11 we can see two relations. T is the result of the Projection of R on the attribute CITY. In the Algebraic Assignment language notation we write

$T := \pi_{CITY}(R)$

R

NAME	CITY
SMITH	LONDON
JONES	PARIS
BLAKE	PARIS
CLARK	LONDON
ADAMS	ATHENS

T

CITY
LONDON
PARIS
ATHENS

Fig. 3.11 Projection

If we assume that R contains the information about customers of some company, then T contains a list of towns where the company has customers.

JOIN

The result of the Join is a relation with tuples concatenated from tuples (or selected attributes of tuples) of the two source relations satisfying some specified predicate e.g. equality of values of specified attributes.

There are many different variants of the Join operation. We will show a Join1, called the natural join in the literature, and a Join2 which is the same as Join1 but with the joining attribute eliminated from the result. The condition for join is the equality between specified attributes.

In Figure 3.12 we can see four relations. R1 and R2 are the source relations and T1 and T2 are the results of the Join1 and Join2 respectively. In the Algebraic Assignment language notation we write

$T1 := R1 \text{ JOIN1 } R2 \text{ OVER } P$

$T2 := R1 \text{ JOIN2 } R2 \text{ OVER } P$

R1		R2	
CITY	P	CITY	P
LONDON	P1	PARIS	P2
ATHENS	P1	PARIS	P1
LONDON	P2	LONDON	P1
BERLIN	P3		

T1			T2	
CITY1	P	CITY2	CITY1	CITY2
LONDON	P1	PARIS	LONDON	PARIS
LONDON	P1	LONDON	LONDON	LONDON
ATHENS	P1	PARIS	ATHENS	PARIS
ATHENS	P1	LONDON	ATHENS	LONDON
LONDON	P2	PARIS		

Fig. 3.12 Join2

Let us imagine that R1 contains information about the location of some parts and R2 contains the information about where those parts should be delivered. By combining the information from R1 and R2 we get by the Join1 and Join2 relations T1 and T2. T1 tells for each part where it comes from and where it will be shipped. T2 gives all pairs of source and destination addresses.

SEMI-JOIN

A Semi-join operation [Bernstein81a] is a variant of a join operation but because of its convenience in evaluating queries, as will be seen in Chapter 6, we define it as a separate operation.

The Semi-join takes two relations, with at least one attribute with values drawn from the same domain, as its arguments. By an intersection between those two attributes this operation gives a result relation which is a subset of the first relation, its tuples pointed out by the attributes of the second relation.

In Figure 3.13 we can see the relation T being the result of the Semi-join of the relations R1 and R2. In the Algebraic Assignment language notation we write

T := R1 SEMIJOIN R2 ON CITY

R1		R2		T	
NAME	CITY	CITY		NAME	CITY
SMITH	LONDON	ATHENS PARIS		JONES	PARIS
JONES	PARIS			BLAKE	PARIS
BLAKE	PARIS			ADAMS	ATHENS
CLARK	LONDON				
ADAMS	ATHENS				

Fig. 3.13 Semi-join

Let us imagine that R1 contains the information about some large group of persons and that we want to restrict this information to persons living in towns listed in R2. T contains this restricted information.

DIVISION

Let us assume for simplicity that the dividend relation has two attributes A and B and the divisor relation one attribute C. Furthermore, B and C have values drawn from the same domain. The result of the Division is a relation which is a subset of the values of the attribute A of the dividend. A tuple x belongs to the result if x is a value of the attribute A of the dividend and for each value y of the divisor the pair x,y is a tuple of the dividend.

In Figure 3.14 we can see three relations, R1 is the dividend, R2 the divisor and T the result of the Division. In the Algebraic Assignment language notation we write:

$T := R1 \text{ DIVIDE BY } R2 \text{ ON PART}$

R1		R2	T
CITY	PART	PART	CITY
LONDON	P1	P1	LONDON
ATHENS	P3	P2	PARIS
PARIS	P2		
LONDON	P2		
PARIS	P1		
BERLIN	P2		

Fig. 3.14 Division

Let us imagine that R1 contains the information about location of some parts, and R2 contains a list of selected parts. T contains names of towns where all parts of R2 are located.

The Division operation is sometimes useful in expressing queries which contain the word 'all'.

Chapter 4.

DATABASE COMPUTERS

The area of research and development of database computers is only about 10 years old. Its emergence is the answer to needs of matching the computer architecture and the functional requirements of data management. In this chapter we will give a brief survey of this area and present some selected hardware approaches.

4.1 CATEGORIZATION OF DATABASE COMPUTERS

Several authors, surveying the fast growing field of database computers have, in order to aid the discussion and provide for comparisons, suggested different classification schemes for them. We will review two categorizations, the first given by Bray and Freeman [Bray79] and the second by Su et al [Su80]. We will also indicate to which category LUCAS belongs if it is used as a database computer.

We will exemplify the two taxonomies by classifying some selected database computers which then will be presented in the next four sections. Some of the computers are commercial products: IDM, iDBP86, CAFS, STARAN, and SYNFOBASE, others are rudimentary prototypes built at universities: RAP and CASSM, and still others are proposed designs (which we call paper machines): DBC, RELACS, and RDB.

In their book Data Base Computers [Bray79], Bray and Freeman classify database computers according to two orthogonal criteria:

- * The number of processing elements involved in the database processing. There is either one processor or a multitude of processors.
- * The type of hardware organization used to search for data. The data are searched either directly on mass storage devices or indirectly in some buffered or intermediate storage area.

The difficulty of defining a workable classification system is illustrated by the fact that Bray and Freeman arrive at five categories instead of four.

1) Single Processor Indirect Search. This category corresponds to the conventional general-purpose computers with firmware and software tailored to a DBMS application. IDM and iDBP86, both commercial computers, belong to this category.

2) Single Processor Direct Search. CAFS, a commercial computer, is a typical proponent of this category. This type of computer implements only some of the many functions of a DBMS. Data are searched by a special-purpose device, making mainly selection and projection operations. Only desired records are sent to the host computer for further processing.

3) Multiple Processor Direct Search. A database is directly searched in parallel by multiple processors attached to the read/write heads of a rotating storage device such as disks, magnetic bubbles or CCDs. Data are read, examined, processed, and written back to memory 'on the fly' during the same revolution. A prototype, CASSM, built at a university, belongs to this category.

4) Multiple Processor Indirect Search. This approach is similar to the previous category. The main difference is that only part of the database is staged and searched in the database computer. To this

category belongs the prototype machine RAP built at a university , the paper machines RELACS and RDM, a commercial product SYNFOBASE, and the famous STARAN. LUCAS, if used as a database computer, also belongs to this category.

5) Multiple Processor Combined Search. The computers in this category combine features of all the previous approaches. Data are searched indirectly in the intermediate medium, but they are also, in a limited way, processed during staging. DBC, a paper machine, belongs to this category.

In a report presented at the 1980 NCC, Su et al [Su80] arrive at four categories of database computers based on a rather intuitive but useful criterion of architectural distinctions and difference in objectives and characteristic. The material in the next four sections is arranged according to this categorization.

1) Cellular-logic systems. CASSM and RAP belong to this category.

2) Backend computers. IDM, iDBP86 and CAFS.

3) Integrated database machines. DBC.

4) High-speed associative memory systems. STARAN, RELACS, RDM and SYNFOBASE. LUCAS, if used as a database computer, belongs to this category.

Any taxonomy for a dynamically evolving area sooner or later becomes unsatisfactory. During the last few years some new designs have been presented which do not fall into any of the categories mentioned above. For example, there are database machines implemented with systolic arrays [Kung80] or tree computers [Song81] taking advantage of a very high degree of integration in VLSI chips.

4.2 CELLULAR-LOGIC SYSTEMS

The database machines in this category consist of a linear array of cells each of which contains a processor and a rotating memory element. The system acts as an intelligent secondary storage device connected to a general-purpose host. Its main purpose is reducing the volume of data brought to the host.

The general architecture of cellular-logic systems is shown in Figure 4.1

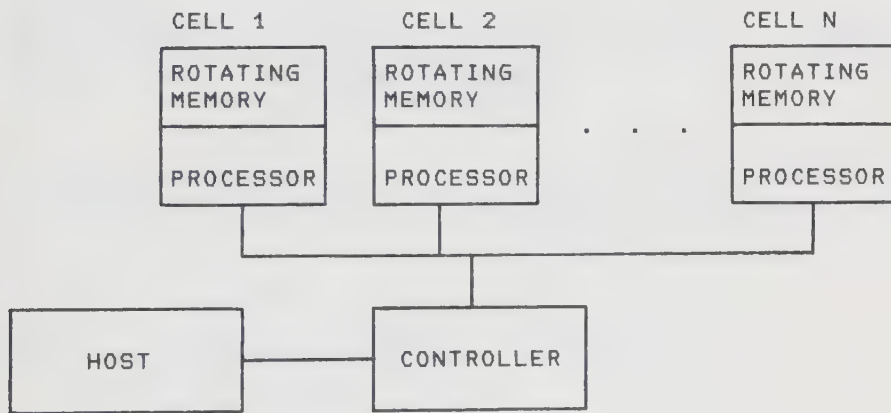


Fig. 4.1 Cellular-logic system

A database operation such as Retrieve, Update, or Delete is broadcast simultaneously to all the processors which carry out the operation on the data residing in their associated memory elements. The memory elements can be any rotating storage medium.

Some of the advantages of this approach are:

- * High processing speed, which is the result of parallelism in operation.
- * Potential for cheap design due to modularity of cells.
- * Use of relatively cheap, serially accessible block memories as associative memories

Typical representatives of the cellular-logic systems are RAP, operating on staged segments of a database, and CASSM containing the whole database in its cells.

RAP

RAP (Relational Associative Processor) [Ozkarahan75] is an experimental cellular processor supporting a relational database model. Its development started in 1974 at the University of Toronto and during the years since then, it evolved through a number of architectural changes. Different generations of RAP called RAP.1, RAP.2 and analyses of different aspects are well documented in the literature [Ozkarahan77a, Ozkarahan77b, Schuster78].

The overall architecture of RAP is shown in Figure 4.2. The machine is composed of three main parts: a Controller, a Set processor, and a parallel array of cells.

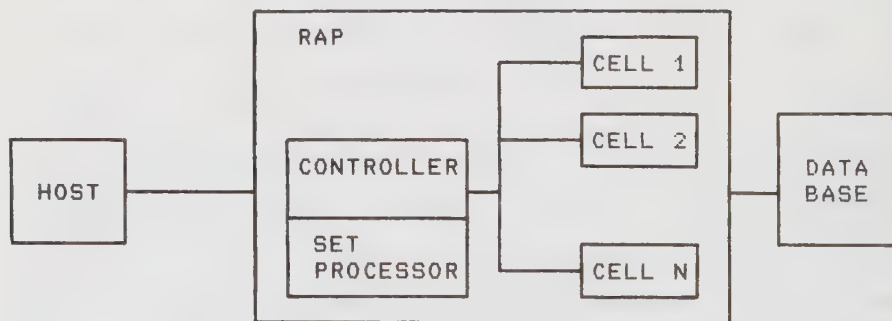


Fig. 4.2 RAP

The host is a general-purpose computer communicating with a user and with RAP. It translates user queries to a sequence of RAP commands and sends them to a RAP Controller. Furthermore, it is responsible for maintaining names of relations and attributes, and their addresses in RAP cells.

The Controller is a processor decoding RAP programs and coordinating overall operation of the cells and of the Set Processor. In RAP.1, the Controller was hardwired, but because of its inflexibility it was replaced by a minicomputer PDP11/10 in RAP.2.

The Set Processor is designed for computing statistics over entire contents of the cells, such as totals, averages, etc.

The cell, depicted in Figure 4.3, consists of a logic component and an associated rotating memory. The logic component can be a microprocessor, the memory can be a track of a disk or drum, or a circular shift register such as a CCD or bubble memory. The cells can communicate with their neighbours.

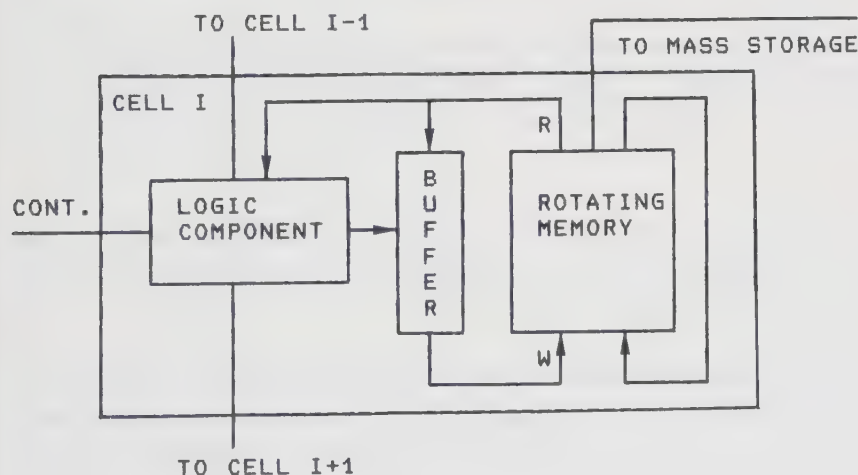


Fig. 4.3 RAP Cell

Data from the rotating memory, read by a read head (R), are copied into a buffer and interrogated by the logic component. The logic component can change the contents of the buffer and as the corresponding record passes a write head, (W), the buffer is copied back to the memory.

The prototype RAP.2 built in 1977 consists of 2 cells each with a 8080 microprocessor as the logic component and 1 Mbit CCD track. A fully-fledged RAP-like database computer would consist of thousands

of cells operating simultaneously on data in their memories.

The most appealing feature of RAP is the simplicity of the representation of a relation in the cells. A circulating memory track in the cell contains a sequence of tuples of the relation, as shown in Figure 4.4

TRACK

RELATION NAME	ATTRIBUTE NAMES	TUPLE1	TUPLE2	...
---------------	-----------------	--------	--------	-----

ATTRIBUTE NAMES

ATTRIBUTE1 NAME	ATTRIBUTE2 NAME	...	ATTRIBUTEN NAME
-----------------	-----------------	-----	-----------------

TUPLE BLOCK

DF	A	B	C	D	LC	ITEM1	LC	ITEM2	...	LC	ITEMN	DL	
----	---	---	---	---	----	-------	----	-------	-----	----	-------	----	--

Fig. 4.4 Memory track of RAP

The first block on the track contains the encoded name of the relation. The second block contains the encoded names of the attributes of the relation.

The remaining blocks contain tuples of the relation. The first bit of the tuple block is a delete flag, (DF), indicating if the following space on the track is occupied by a tuple. The next four bits, (A,B,C,D) are mark bits used by the microprocessor as a work area to temporarily indicate subsets of tuples during the computation. In front of each data item there is a two bit length code, (LC), telling whether data are represented by 8, 16, or 32 bits. The last item in the block is a delimiter, (DL).

RAP can perform a number of machine instructions from which complex database processing programs are constructed. The use of mark bits makes it possible to utilize the result of one instruction in subsequent instructions.

The general format of most RAP instructions is:

<opcode><mark option>[<object>:<qualification>]

The opcode specifies the data manipulation operation, the mark option specifies which mark bits are to be set or reset, the object specifies relations or cells, and the qualification is a conjunction or disjunction of conditions on values of attributes or mark bits.

The following instructions illustrate the potential of RAP:

Select Mark (AB) [R1: D1 = '1'], will set to one the mark bits A and B of those tuples in R1 which have the value of the attribute D1 equal to 1. This instruction is performed during one revolution of the data track.

Cross-Select Mark (A) [R1: D1 = R2.D2], will set to one the mark bit A of those tuples in R1 which have the value of the attribute D1 equal to the value of the attribute D2 in R2. The execution of this instruction requires a number of revolutions equal to 1 plus the cardinality of R2.

Delete [R1: A = 1], will delete those tuples of R1 which have the value of the A mark bit equal to one. One revolution is necessary for the execution.

Compared to conventional systems, by an analytical performance evaluation [Ozkarahan77], RAP achieves advantages in speed between one and three orders of magnitude.

CASSM

CASSM (Context Addressable Segment Sequential Memory) [Su75, Lipovski78, Su79, Hong81] was developed at the University of Florida. It was designed primarily for using a hierarchical data model, which was linearized in a top-down, left-to-right manner, although a relational data model can also be supported. The work on the project began in 1972 and evolved during the following decade. A prototype with one cell was built, using a fixed-head disk as data storage device, and a number of different algorithms were designed

and tested on it. Later on a multicell configuration was simulated and the performance of different configurations were studied.

CASSM contains a linear array of cells, similar to RAP, each containing a processing element and a circular memory element. The memory element can be a track on a disk, a bubble memory or a CCD. Figure 4.5 shows the overall organization of CASSM.

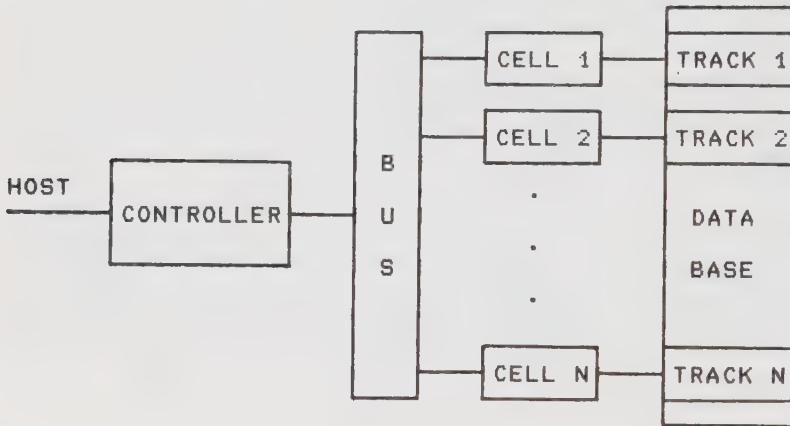


Fig. 4.5 CASSM

Data as well as compiled programs are stored in the memory elements and are scanned and manipulated by the associated processing elements. The assumption is made that the entire database can reside in the storage. A database is a collection of sequential files stored serially, starting from the first element to the n -th memory element. Physically, files are partitioned into segments residing in the memory elements. The segments are simultaneously processed by all the processing elements, thus dividing the total processing task into n concurrent subtasks each involving $(1/n)$ -th of the database. Data or instructions are read by a read head from the circular memory, processed by the processing elements and written back to the memory element by a separate write head. The processing elements can communicate with their neighbours.

Each file is logically a collection of records containing hierarchically structured data. A data word consists of 40 bits, 32 of which are data bits and remaining 8 are tag and status bits. Three tag bits are used to identify the seven word types. The word types include data, instruction, pointer, delimiter, and others. Data can be searched associatively, by content, or by the use of pointers.

The processing elements consist of a number of simultaneously operating modules organized in a pipeline. The fact that instructions are stored in the same memory as data and are fetched for execution by the processing elements makes CASSM independent of the host computer during the execution of a program. The execution of an instruction proceeds in three phases. In the first phase, the next instruction and its operands are located, fetched and sent to all the cells. During the second phase, all the cells execute the instruction against the entire database. In the third phase, the instruction is deactivated.

CASSM is primarily a research tool and only some DBMS functions are implemented.

4.3 BACKEND COMPUTERS

Existing backend computers in database systems are dedicated general-purpose computers for carrying out DBMS functions. High performance is achieved mainly by the use of specialized software. Usually, one backend computer may serve several hosts.

IDM

IDM (Intelligent Database Machine) [Epstein80, Moulds82], manufactured by Britton-Lee Inc., is a relational database computer containing a complete database management system. It is currently marketed in two models: IDM 500, and its stripped-down version IDM 200, each with many different options.

IDM can serve as a stand-alone device supporting several intelligent

terminals or as a centralized database resource for up to 16 host mini- or microcomputers. A database which it controls may contain thousands of relations with hundreds of attributes and totally a few billions of tuples in total.

The base configuration, shown in Figure 4.6, consists of a database processor, implemented with a Zilog Z-8000 16-bit microprocessor, a 256 kbyte RAM, a disk controller supporting four disks, and a serial or parallel I/O channel.

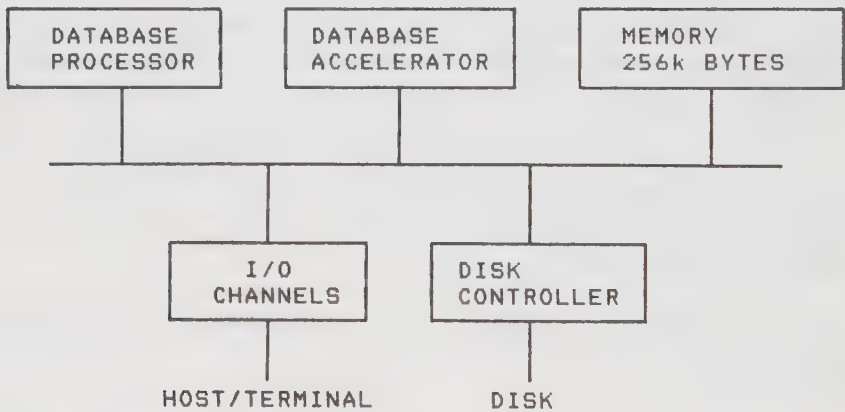


Fig. 4.6 IDM

IDB may also include a special Database Accelerator, sold as an option. It increases the system performance by a factor of 30. The Database Accelerator is a special purpose processor built from Shottky TTL logic, executing a carefully selected collection of subroutines in microcode very rapidly.

The main application area for IDM is to work together with a minicomputer in small business, for example in CAD/CAM design or in an automated office environment.

iDBP 86/440

iDBP 86/440 (Intel Database Processor) [Hughes82], is manufactured by the Intel Corporation as an OEM product. It is a microprocessor-based, relational database management system functioning as an intelligent mass storage controller for one or more hosts. It acts as a slave to the host, performing database functions as instructed.

The hardware is basically a multiprocessing system utilizing two 16-bit, 8 MHz 8086 microprocessors. One microprocessor handles communication with the hosts, the other executes all DBMS functions and manages various hardware resources. Furthermore, there are Intel 8089-based mass storage controllers supporting up to 16 standard disks. The size of the primary memory is maximum 1 Mbyte.

System capacity is a maximum of 255 databases with a maximum of 255 relations (files) each. A tuple (record) can have a maximum of 127 attributes.

The system software provides relational operators like Join, Projection, and Selection. For applications designed for hierarchical or network architectures it provides tools to implement pointers and many-to-many relationships.

iDBP can also manage files of unstructured information, making applications like word processing or processing of digitized image and digitized voice possible.

CAFS

CAFS (Content Addressable File Store) [Maller79] is a product of International Computers Limited (ICL). It is a high speed information retrieval system for accessing data by their attributes and values, based on intelligent hardware running as a backend processor to a host which is an ICL mainframe. It is strictly a retrieval system not allowing on-line updates.

CAFS can access up to 840 Mbytes of formatted records stored on

standard ICL disks. Each record consists of a string of self-identifying fields of variable length. Each field identifier defines the type of data contained in the field, e.g. name, supplier number, etc.

Figure 4.7 shows the overall system organization of CAFS. It consists of a Control Computer (which is an ICL minicomputer), a Selection Unit, a Retrieval Unit, and a Direct Access Unit.

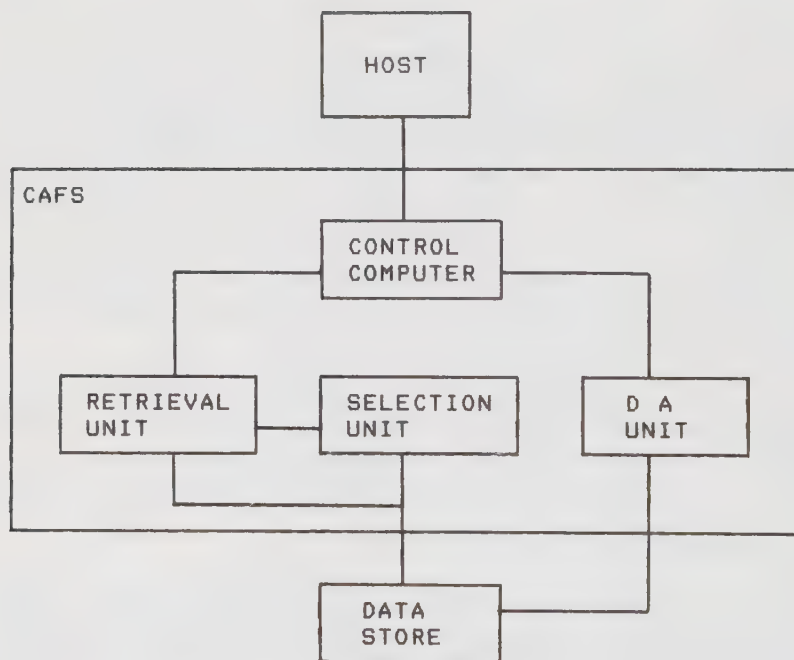


Fig. 4.7 CAFS

The Retrieval Unit comprises a number of data buffers which are continuously filled with passing data from the data store according to the user's retrieval specification. If the Selection Unit indicates that the data in the Retrieval Unit satisfy the query, the data are sent to the Control Computer and from there to the host. Otherwise they are discarded.

The Selection Unit is the heart of the system. It contains four subunits: a Data Multiplexing Unit, a Search Evaluation Processor, a Key Channel Unit, and a File Correlation Unit.

The Data Multiplexing Unit takes data from up to 12 individual disk channels and produces a single multiplexed output on a byte wide highway. The Key Channel Unit contains 16 key and mask registers together with corresponding comparators. The comparators, operating simultaneously on the data stream, detect the presence of a key type, and the equivalence (or the inequalities 'less than', 'greater than', etc) of key data and the value of an attribute. The result of a comparison is used as an operand in a microprogram executed by the Search Evaluation Processor. The microprogram is compiled by the host from the user's selection expression.

The File Correlation Unit enables queries to be evaluated on joins of two physical files without sorting or merging intermediate results, provided that both share the same attribute. The unit operates with the help of a set of 256k 1-bit wide stores addressed by the values of the joining attribute.

The operation of the File Correlation Unit proceeds in the following way. First, one file is processed by the Search Evaluation Processor, which sets bits in the 1-bit wide store corresponding to the values of the attribute satisfying the search. On a subsequent search on the second file by the Search Evaluation Processor the store is treated as an output of the Key Channel Unit, thereby enabling a linked selection operation to be carried out across the two files.

The Direct Access Unit provides conventional facilities for disk initialization and block read/write.

The main application for CAFS is information search in large databases, e.g. telephone directory enquiries, bibliographic retrieval, personal credit control etc.

4.4 INTEGRATED DATABASE MACHINES

The database computers in this category consist of cooperating, functionally specialized units implementing different functions of a DBMS. They can even contain as a component a cellular logic computer or an associative memory system.

DBC

DBC (Data Base Computer) [Banerjee78, Banerjee79] is a paper machine developed at the Ohio State University. There were five major objectives of the design: 1) Capability to handle very large databases of 10 to 100 billion bytes. 2) Use of current (mid 1970s) technology. 3) Performance/cost much better than existing DBMS implemented on general-purpose computers. 4) Inclusion of security mechanism for control of sharing and protection. 5) Possibility of supporting all important data models.

DBC acts as a back-end to one or more hosts. User programs reside in the host, and are executed by the host which uses DBC as an attached resource. The host sends commands to DBC consisting of the operation code, one or more keyword predicates and the data items. DBC responds either by returning a group of demanded records or parts of records, or by indicating successful or unsuccessful execution of an update command.

DBC consists of seven major elements, some of which are associative processors. The elements are organized into two loops: a structure loop and a data loop. The operation of loops is pipelined. The basic architecture is shown in Figure 4.8.

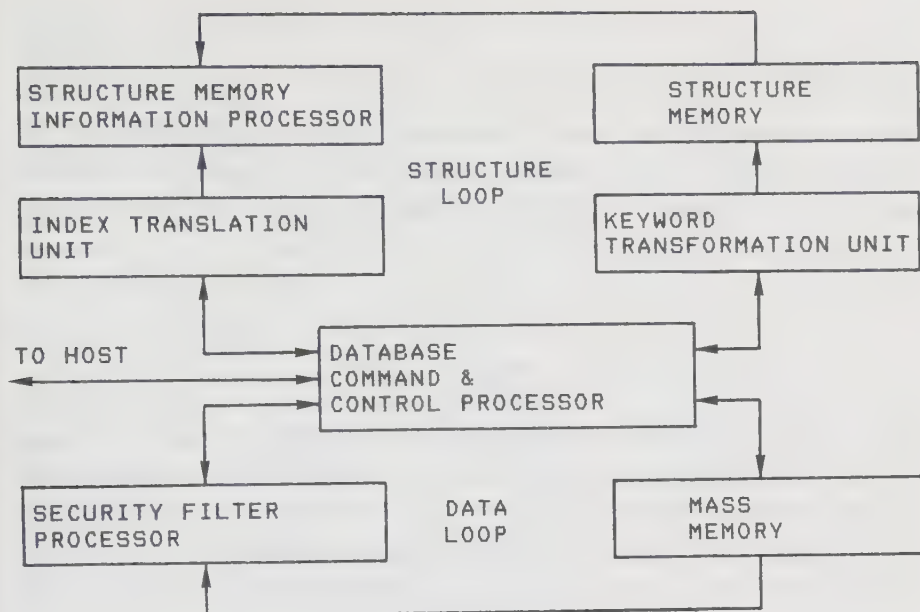


Fig. 4.8 DBC

The Database Command & Control Processor receives user requests from the host and converts them into instructions to other DBC elements.

The data loop is used for storing and accessing the database and post processing of retrieved records. The database consisting of records is stored in the Mass Memory which is implemented with moving-head disks. The moving-head disk drives are modified to provide for simultaneous output from all tracks in a given cylinder in parallel. With each track there is a track information processor which performs associative comparison operations.

Because DBC is intended to be used for very large databases, which are impossible to search sequentially, some structural information about the database must be used. The structural information is stored in a Structure Memory. The size of the Structure Memory is about 1 percent of the size of the Mass Memory. The function of the

structure loop is to translate the selection expression from the host into physical addresses to blocks of data on the Mass Storage.

With a large number of common database functions implemented in the hardware, DBC has an estimated processing power 80 to 160 times that of a conventional DBMS on a general-purpose computer with all functions provided by the software.

4.5 HIGH-SPEED ASSOCIATIVE MEMORY SYSTEMS

The database computers in this category use an associative memory for high-speed searches by content or context on data loaded from conventional memory devices such as disks. There are two aspects that make the associative memory particularly attractive for use in database management: content addressing and parallelism in operation.

Unfortunately there is also a limiting factor, which is input and output of data. If loading the data takes much longer time than the parallel processing then the performance advantage of using the associative memory would be eliminated. This problem was studied extensively by Berra [Berra79], who demonstrated how it can be overcome by the use of a high band-width interface between the associative memory and a fast secondary staging memory.

A typical system configuration is shown in Figure 4.9.



Fig. 4.9 Associative Memory System

STARAN

STARAN [Batcher74, Gambino75] was built as a commercial product by Goodyear Aerospace Corporation in the mid 1970s. Until now, it is the most well-known associative computer ever built. Originally, it was designed for image processing in military applications, but because of its associative hardware it can be used in a database computer. Its applicability to a database management system was studied at Syracuse University [DeFiore74].

Figure 4.10 shows the overall organization of STARAN. The major components are: a Sequential Controller implemented with a PDP-11 minicomputer, a Control Memory with microprograms, a microprogrammed Associative Array Controller, an External Function Logic synchronizing the operation of other parts and communicating with a host, and an Associative Array.

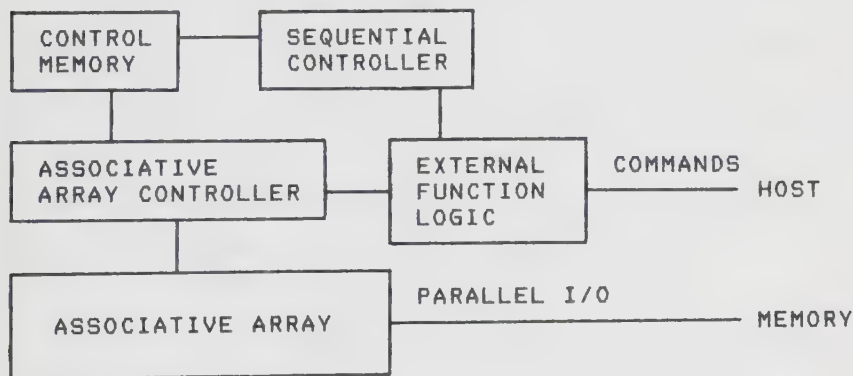


Fig. 4.10 STARAN

The Associative Array, somewhat similar to the Associative Array of LUCAS, consists of 1 to 32 arrays. In the original STARAN B, the arrays were constructed from 256, 256 bit random access memories. These were the largest RAM devices available at that time. Since that time, the STARAN E has been developed taking advantage of the

improved memory technology. The array modules are constructed from 1024 bit bipolar RAM devices and 4096 bit MOS RAM devices. Addressing in an array is in word mode or in bit-slice mode. With the aid of a flip network there are also other addressing modes but they are of minor interest in database management applications. A maximum of 256 bits may be accessed at a time. With each word there is a simple ALU with three flip-flops, M, X, and Y. The X flip-flop is a temporary fast storage participating in logical operations as both source and destination. Other participants in the operations are the Y flip-flop and a bit from the memory word. The M flip-flop is used as a mask register for words and a word-select register in the writing of bit-slices into the memory.

Control instructions are broadcast to the Associative Array in parallel by the Associative Array Controller and they are executed simultaneously in all words. The mode of operation is bit-serial, word-parallel. All arrays share a register in the Associative Array Controller providing for a single compare operation in parallel in all memory words.

A typical system configuration, using STARAN, is shown in Figure 4.11.

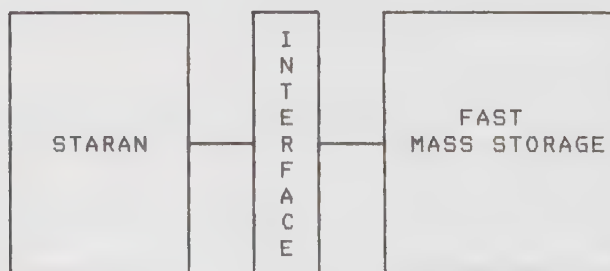


Fig. 4.11 STARAN system configuration

A database stored on a fast mass storage is partitioned into pages of the same size as the Associative Array. The search procedure over the entire database consists of 1) location of a page in the mass storage, 2) moving the page across the fast interface to STARAN,

and 3) parallel search in the array. Berra [Berra79] estimates that the time to search a database of 128 Mbytes in a four array STARAN is two seconds.

Moulder [Moulder73] developed an experimental, research oriented, hierarchical database management system utilizing STARAN and a parallel head-per-track disk drive with a Sigma 5 general-purpose computer as a host. Retrieval of data was performed by selection based on equality over various attributes. The database was partitioned into a number of physical disk segments which were successively read into the Associative Array in parallel fashion.

STARAN and LUCAS have many features in common. The STARAN architecture had great influence on the design of LUCAS.

RELACS

RELACS (Relational Associative Computer System) [Oliver79] is a paper machine developed at Syracuse University. It is a backend database computer designed to support a relational data model. The main goal of the design was to achieve the highest possible performance with maximum possible use of new hardware techniques, particularly associative memories. The associative memories are utilized for search operations in data dictionary and directory functions, language translation functions and database search operations implementing relational algebra.

The RELACS system configuration is shown in Figure 4.12. There are six main functional units: a Global Control Unit (GCU), a Dictionary/Directory Processor (DDP), an Associative Query Translator (AQT), Associative Units (AU0, AU1), a Mass Storage Device (MSD), and an Output Buffer (OB).

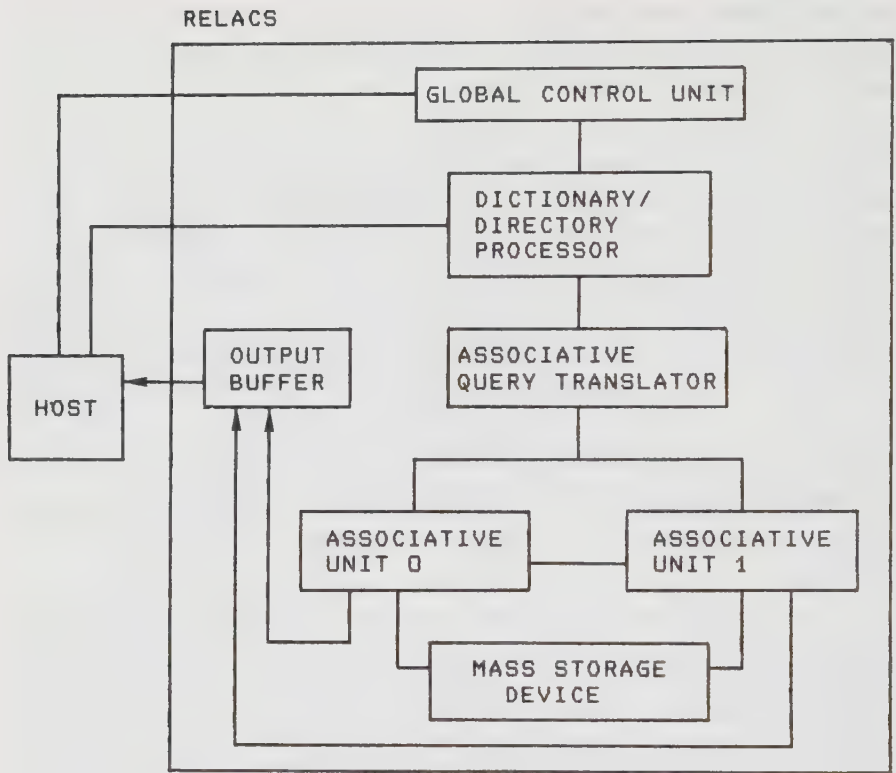


Fig. 4.12 RELACS

The database resides in the MSD with an assumed capacity of over a billion bits. It is assumed that data can be transferred to the AUs at a speed compatible with the search speeds of the units. Queries to the database are preprocessed by the host which performs syntactical functions to assure that the query is syntactically correct.

The DDP establishes that the user has a correct password, that the relations and attributes exist and that the logical relationships necessary for answering the query can be constructed. It outputs to the AQT the descriptor data for all attributes referenced directly and indirectly by the query. The DDP provides the capability to

precisely locate the data which must be searched in a process of query evaluation. The AQT uses these descriptor data together with the original query to create a set of instructions executable by the AUs.

The most interesting part of the AU is a multiple comparand register giving it the capability for many to many comparison, speeding up the join operation.

RELACS was comparatively evaluated with the RAP system for retrieval and update for expected relation cardinality of 10^5 . For simple queries, RELACS was faster by a ratio of 2 to 75, for complex queries the ratio was between 20 and 3500.

RDB

RDB (Relational Database Machine) is a paper machine, developed by D. E. Shaw [Shaw79] at the Stanford University. The architecture is organized as a two level hierarchy of associative storage devices.

The organization of RDB is shown in Figure 4.13. RDB consists of three main components: the Host which is a general-purpose computer, the Primary Associative Memory (PAM), and the Secondary Associative Memory (SAM).

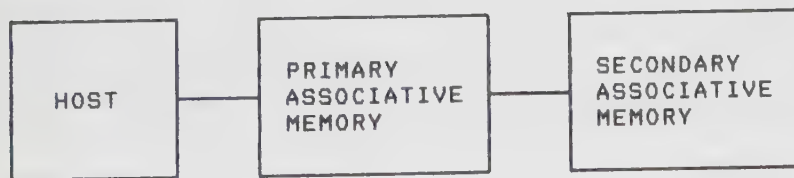


Fig. 4.13 RDB

The PAM is a fast associative memory of rather limited size of between 10k and 1M bytes and a speed of between 100 nanoseconds to 10 microseconds for one search. It can perform two primitive operations: Mark All, and Retrieve and Mark First. In both cases all tuples of a relation in PAM with the value of a selected attribute

equal to a particular constant are associatively identified. The Mark All operation writes in parallel a one or zero in a specified flag bit of each matching tuple. The Retrieve and Mark First operation sets a specified flag bit within a single tuple chosen arbitrarily among the responders and copies the value of that tuple to an external register accessible by the Host.

The SAM is a block-oriented associative storage device much larger but slower than PAM. It has a capacity of between 1 and 100 Mbytes. The SAM can be realized using a parallel head-per-track disk or CCD with a simple processor, similar to the one of RAP or CASSM, associated with each storage loop. The processor has the capability of examining an attribute in each tuple passing a read head and collecting such matching tuples for output. Furthermore, it can compute a simple hashing function of the attribute in question and output all tuples for which the resulting hashed value falls within a specified range.

The motivation for including a hierarchy of two associative devices with different capabilities in RDP, is the observation that large, fast associative memories are too expensive. Shaw has concentrated on demonstrating algorithms for a number of relational algebraic operations, e.g. Projection and Join, in the case where the argument relations exceed the capacity of PAM. The combination of PAM and SAM permits an $O(\log n)$ decrease in execution time compared to conventional evaluation methods on von Neumann type computers. If compared to RAP or CASSM the performance is improved by a factor proportional to the size of PAM.

SYNFOBASE

SYNFOBASE [Lamb82] is a commercial product which was developed by AEG-Telefunken in Germany. It is a small scale database computer which contains an associative processor. SYNFOBASE can be connected to any microcomputer or minicomputer host by means of a cable with a serial interface. The intended application is to provide users of small computers with a fast intelligent data bank.

The organization of SYNFOBASE is shown in Figure 4.14. SYNFOBASE consists of the following components connected to an STD bus: a Z-80 CPU, a RAM of 56 kbyte, a Winchester disk drive and disk controller, a DMA controller, a floppy-disk drive for back-up, and a REM (Recognition Memory) associative processor.

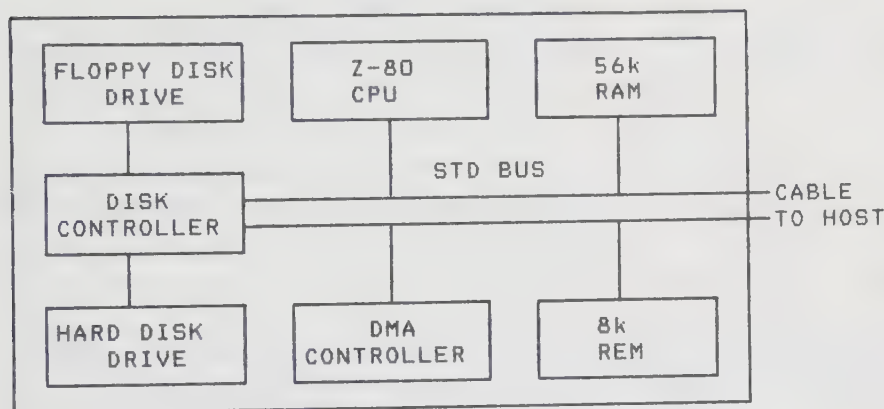


Fig. 4.14 SYNFOBASE

The heart of the SYNFOBASE is the REM developed by Semionics Associates [Lamb78]. The REM is an 8 kbyte memory module organized into 32, 256 byte long, "superwords". With each of the superwords there are two flip-flops called T and S. The T flip-flop is usually set by a successful comparison operation between a byte in the superword and a byte on a data bus. The T flip-flop may be used for selective control of writing a byte into the corresponding superword. The S flip-flops have a variety of functions, mostly concerned with saving the existing values of the T flip-flops.

The internal operation of REM is byte serial, and there is only one processor which operates on bytes. But because this operation is very fast, from the outside world it looks as if the REM was operating simultaneously on one byte across all 32 superwords at a time. (This feature makes the REM hardware much less expensive than if it were operating in a truly parallel fashion.) There are 17 primary operations provided by the REM, including seven types of

compare (e.g. equality, less than, etc), four types of multi-write (e.g. write all, write if T=1, etc), and six operations on the T and S flip-flops (e.g. OR, exchange, etc). If an associative operation is performed on a string of bytes it will execute by byte. The REM also includes a mask register providing for bit operations.

SYNFOBASE contains only one REM, which is obviously too small for most applications. Larger files are run through the REM in 8 kbyte blocks from the Winchester disk. The very efficient disk controller and DMA make the REM and the disk act as a virtual large REM. The capacity of the REM-load is the same as that of one track on the disk. For a typical search, the time required for processing in REM is so short that it is ready for more data as soon as the disk is ready to start reading the next track.

Since REM can find substrings in arbitrary positions very rapidly, SYNFOBASE is also useful in environments with unstructured databases.

Chapter 5.

RELATIONAL ALGEBRA ON LUCAS

This chapter presents the implementation of relational algebra operations on LUCAS. Some of the results of this chapter were presented at the Sixth Workshop on Computer Architecture for Non-Numerical Processing, Hyeres, France, June 1981 [Kruzela81].

We will demonstrate algorithms and derive their exact timing equations. We assume that the size of the relations is less than the size of the Associative Array.

The purpose of the presentation is twofold:

- * To show that the Associative Array can be used for more than just searching data in the context of relational database management.
- * To assess the efficiency of the implementation of the algebraic operations. ig

The timing equations are most helpful in analyzing the performance of the Associative Array. The equations will express the total execution time in terms of number of clock cycles consumed by the execution. Parameters in the timing equations will be the size of tuples or attributes and the cardinality of involved relations.

The operations are implemented by microprograms which are initiated

by the Supporting Processor. Prior to any operation, the Supporting Processor must send all the necessary parameters e.g. the addresses of attributes or the size of tuples, to the Control Unit of the Associative Array. The parameters are stored in registers of the Address Processor.

The level of detail in the presentation of algorithms is somewhere between a plain listing of annotated microprograms and a description in the high level notation of Pascal/L [Fernstrom82]. The listing of microprograms would make it easy to determine the timing equations but all the tedious technical details would obscure the principles behind the implementation. (Writing microprograms gives rich opportunities for efficient but seemingly inexplicable solutions. Reading microprograms is never simple.) Pascal/L notation would make the principles of implementation clear, but it would be less useful in explaining how algorithms are actually implemented and would be useless for deriving the timing equations.

We will use a simple, largely self-explanatory notation. Details of the implementation without principal interest e.g. how the Address Processor controls loops or which internal registers are used, will be omitted. In tables, giving a step by step description of the implementation, we always indicate in the rightmost column the timing of each step. For comparison, we will show real microprograms in a few cases.

To facilitate the understanding of some of the operations, we will give simple diagrams showing the state of the Associative Array before and after the operation and in some cases also during execution of the operation. The diagrams are based on the schematic picture of the Associative Array shown in Figure 5.1. Only the particular section of the Associative Array which is relevant for the operation will be displayed.

I/O Buffer Register



I0B

I/O Register



I0

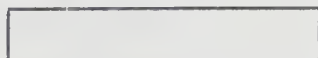
Comparand Register



I/O Register



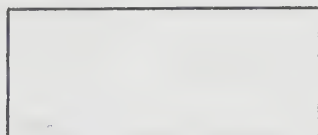
Mask Register



I/O Registers



Memory Array



X

C

R

T



Fig. 5.1 Schematic picture of Associative Array

In the diagrams we will use the letters S and D above a box representing the Memory Array to indicate the source and destination of data; 0 and 1 denote a value of a bit; x is an unspecified value of a bit; A,B,E stand for a value of a byte; and X (in the Memory Array) is an unspecified value of a byte.

In the next section we will show how a relation is represented in the Memory Array.

In Section 5.2 we will describe the implementation of some basic operations in the Associative Array. We will first give algorithms for some very simple operations on bitslices. Then, we will gradually increase the complexity of operations, bitslices - byteslices - words - fields. In describing new operations we will use those previously defined. In Section 5.3 we will arrive at a description of the implementation of algorithms for the operations of relational algebra. Finally, in Section 5.4 we will summarize and discuss our results.

5.1 REPRESENTATION OF A RELATION IN THE ASSOCIATIVE ARRAY

There is an obvious mapping between the logical structure of a relation and its physical representation in the Memory Array. The relation is a table, and the most straight forward way to store it in the Memory Array, which is also a table, is to allocate each tuple to one memory word, so that the attributes occupy vertical fields in the array.

Figure 5.2 shows as an example, a relation consisting of four attributes stored in the Memory Array.

MA

S1	SMITH	20	LONDON
S2	JONES	10	PARIS
S3	BLAKE	30	PARIS
S4	CLARK	20	LONDON
S5	ADAMS	30	ATHENS

Fig. 5.2 A relation in the Memory Array

A relation in the Memory Array is identified by two sets of parameters:

- * Information about which memory words hold its tuples.
- * The size and addresses of its attributes.

The information about which memory words hold tuples of the relation is stored in the Memory Array together with the relation. With each relation in the array there is a unique byteslice called a Workfield at an address assigned by the Supporting Processor. One particular bitslice in the Workfield indicates by a 1 in its bit pattern that the corresponding memory word holds a tuple of the relation. This bitslice is called a Markbitslice of the relation. The other bitslices

of the Workfield are used as a scratch pad during the execution. The content of the Workfield is invisible to the Supporting Processor. Since all operations on data in the Memory Array are always performed in parallel and data are accessed associatively there is no reason why the outside world should know in which memory words the relation is stored. Before operating on the relation the Markbitslice is usually loaded into the Tags.

The address of an attribute of the relation in the Memory Array is a 12-bit bitaddress (0 . . . 4095) to its rightmost bitslice. The addresses of relations currently in the Memory array are maintained by the Supporting Processor. They are assigned to the relation when it is loaded into the Memory Array or when it is created as a result of some operation on relations already in the Memory Array. The Supporting Processor keeps track of a pool of free space in the Memory Array. Before a new relation is to be loaded into the array, or before a new relation is created from relations already existing in the array, the Supporting Processor checks the size of the attributes (number of bytes) and assigns proper addresses to them. It also assigns an address to the Workfield.

Since addressing of bitslices in the Memory Array is made by random access, any two bitslices may be logical neighbours. The attributes of the relation do not need to occupy a contiguous field in a memory word. The order between the attributes is arbitrary. Furthermore, the attributes of different relations may be freely intermingled in one memory word.

Figure 5.3 shows two relations, S and J, residing simultaneously in the Memory Array. The relation S has three attributes with addresses SA1, SA2, SA3 and a Workfield at address SWF. J has four attributes with addresses JA1, JA2, JA3, JA4 and a Workfield at address JWF. The figure displays only the content of the Markbitslices.

	SA1	JA1	JA3	SA3	SA4	JA2	SA2	SWF	JWF
MA									

S1	J1	PARIS	20	LONDON	SORTER	SMITH	1	1
							0	0
S2			10	PARIS		JONES	1	0
							0	0
S3	J2	ROME	30	PARIS	PUNCH	BLAKE	1	1
	J3	ATHENS			READER		0	1
S4	J4	ATHENS	20	LONDON	CONSOLE	CLARK	1	1
S5			30	ATHENS		ADAMS	1	0
	J5	LONDON			COLLATOR		0	1

Fig. 5.3 Two relations in the Memory Array

One item of data physically represented in the Memory Array can belong to many different relations. This is frequently the case when a query to a database is evaluated inside the Memory Array, as we will see in Chapter 6. In Figure 5.4, we can see four different relations S, T, Q and R. S is the original relation loaded into the Memory Array, T is the same as S with the only difference that the values of the fourth attribute of T consist of only three letters, Q consists of the tuples of S whose fourth attribute has value Paris, R is the result of the Projection on S over the fourth attribute.

S:	SA1	SA2	SA3	SA4	SWF				
T:	TA1	TA2	TA3	TA4		TWF			
Q:	QA1	QA2	QA3	QA4			QWF		
R:				RA1				RWF	
MA									

S1	SMITH	20	LONDON	1	1	0	1
S2	JONES	10	PARIS	1	1	1	1
S3	BLAKE	30	PARIS	1	1	1	0
S4	CLARK	20	LONDON	1	1	0	0
S5	ADAMS	30	ATHENS	1	1	0	1

Fig. 5.4 Four relations in the Memory Array

5.2 BASIC OPERATIONS IN THE ASSOCIATIVE ARRAY

Algorithms operating on relations in the Associative Array can be naturally decomposed in a repeating sequence of basic operations. In the next three sections, we will describe the implementation of some of the basic operations and we will also give their timings.

The basic operations will operate on the following items of data:

- * A bit.
- * A byte.
- * A bitslice - one bit at the same bitaddress in all words of the Memory Array.
- * A byteslice - one byte at the same address in all words of the Memory Array.
- * A word - a consecutive sequence of bytes in one memory word.
- * A field - a number of words in the Memory Array at the same address.

5.2.1 BITSlice OPERATIONS

The bitslice operations are used for:

- * Loading bitslices from the Memory Array into flip-flops of the corresponding processing elements.

- * Storing values of the flip-flops in the Memory Array.
- * Moving a bitslice from one bitaddress in the Memory Array to another.
- * For simple processing of bitslices.

The efficient implementation of bitslice operations is very important because they are included in most of the more complex operations.

In this section we will only describe some of the more typical operations.

1) Load bitslice operations

There are four types of load operations:

- * LOAD R
- * LOAD RCTJ, shown in Figure 5.5
- * LOAD T
- * LOAD TCTJ

One bitslice from the Memory Array, at an address supplied by the Address Processor, is loaded into the R or T flip-flops. The state of the T flip-flops (Tags) may be used for selective control of the execution of the operation. The execution will be performed only in those processors in the Associative Array where the Tags are set to one.

LOAD R	:	MA(S) --> R	(1)
LOAD RCTJ	:	MA(S) --> RCTJ	(1)
LOAD T	:	MA(S) --> T	(1)
LOAD TCTJ	:	MA(S) --> TCTJ	(1)

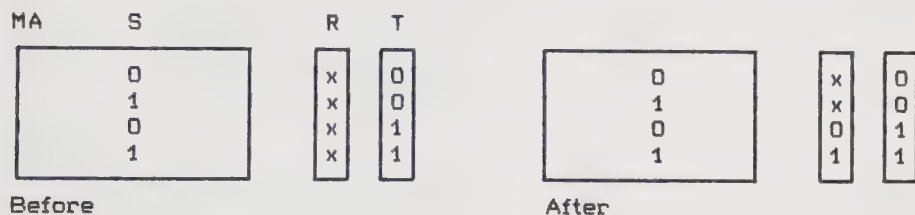


Fig. 5.5 LOAD RCTJ

Each of the four load bitslice operations are executed in one clock cycle.

2) Store bitslice operations

There are two types of store operations:

* STORE R, shown in Figure 5.6

* STORE RCTJ

In both operations, the contents of the R flip-flops are stored in the Memory Array at an address supplied by the Address Processor. In the STORE RCTJ operation, the values of the R flip-flops are stored only in the memory words where the Tag is set to one.

STORE R	:	R --> MA(D)	(1)
STORE RCTJ	:	R --> MA(D)CTJ	(1)



Fig. 5.6 STORE R

Both store bitslice operations are executed in one clock cycle each.

As a matter of fact, it is the STORE RCTJ operation that makes LUCAS a content addressable computer. The state of the Tags may be the result of some previous search operation over all words in the Memory Array. The Tag was set or reset if the content of the corresponding word satisfied or dissatisfied the search criterion. The STORECTJ is then performed in the memory words selected according to their contents.

3) Moving a bitslice

A bitslice may be transferred from one bitaddress to another by operations:

- * MOVE BITSLICE, in all words of the Memory Array, shown in Figure 5.7
- * MOVE BITSLICECTJ, in the Memory words where the Tags are set to one.

MOVE BITSlice	:	1) LOAD R	;S	(1)
		2) STORE R	;D	(1)
MOVE BITSlice[CT]	:	1) LOAD R[CT]		(1)
		2) STORE R[CT]		(1)

MA S D

0	x
1	x

0	0
1	1

Before

After

Fig. 5.7 MOVE BITSlice

These operations are executed in 2 clock cycles each, one for loading the bitslice into the R flip-flops and one for storing it back to the Memory Array.

4) Logical operations on bitslices

Slightly more complex than the previous operations are logical operations on bitslices in the Memory Array. Typical operations are:

- * AND
- * OR
- * XOR (exclusive or)

All operations can be performed in all memory words or only in those words where the Tags are set to one.

We only demonstrate the logical operation AND, shown in Figure 5.8. The other logical operations are quite similar. The AND is executed in 3 clock cycles. In the first clock cycle, a bitslice (S1) is loaded into the R flip-flops from the Memory Array, in the second clock cycle, AND is performed between the R flip-flops and another bitslice (S2) from the Memory Array, with the result loaded into the R flip-flops. In the third clock cycle, contents of the R flip-flops are stored into the Memory Array (D). Addresses to bitslices are supplied by the Address Processor.

AND	1) LOAD R	;S1	(1)
	2) R AND MA(S2) --> R	;S2	(1)
	3) STORE R	;D	(1)

MA S1 S2 D

0	1	x
1	1	x
1	0	x
0	0	x

Before

0	1	0
1	1	1
1	0	0
0	0	0

After

Fig. 5.8 AND

The logical operations are all executed in 3 clock cycles.

5) Selecting the next valid word

In many complex algorithms, there is a situation when data in a number of memory words must be processed sequentially, word after word. A bitslice in the Memory Array, called a Markbitslice, indicates in which words the valid data are stored. For sequential processing we need a mechanism for selecting the first memory word according to the information in the Markbitslice and also for resetting the corresponding bit in the Markbitslice to indicate that the word was chosen (removed from the list).

The operation

* SELECT FIRST AND REMOVE

shown in Figure 5.9 implements this function.

The execution proceeds in the following way:

In the first clock cycle the Markbitslice is loaded into the Tags. In the second clock cycle, the operation Select first is performed on the Tags, resetting to zero all Tags except the first. In the third clock cycle, the NONE signal, indicating whether any of the Tags are set to one, is tested by the Microprogram Controller in the

Control Unit. If none of the Tags are set to one the operation is aborted, otherwise the contents of the Tags are copied into the R flip-flops. In the fourth clock cycle the logical operation XOR is performed between the R flip-flops and the Markbitslice in the Memory Array with the result saved in the R flip-flops. Finally, in the fifth clock cycle, the R flip-flops are stored in the Memory Array. The effect of this operation is that the first Tag according to the Markbitslice is set to one and the Markbitslice is updated.

SELECT FIRST AND REMOVE:	1) LOAD T		;M (1)
	2) SELECT FIRST		(1)
	3) T	--> R	(1)
	EXIT IF NONE		
	4) MA(M) XOR R	--> R	;M (1)
	5) STORE R		;M (1)

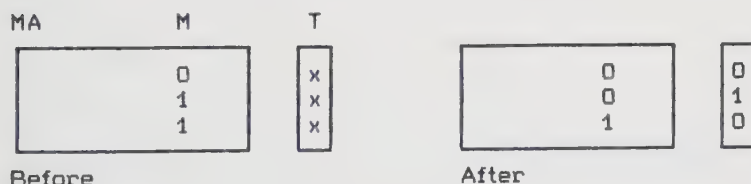


Fig. 5.9 SELECT FIRST AND REMOVE

The execution of this operation takes five clock cycles.

5.2.2 BYTE TRANSFER OPERATIONS

The data item to be transferred in this group of operations is a byte or a byteslice.

1) Transferring a byteslice between the I/O Registers and the Memory Array.

The contents of the I/O Registers are transferred in parallel to all memory words in the Memory Array by the operation

* STORE IO

or only to the memory words with the Tag set to one, by the operation

* STORE IOCTJ, shown in Figure 5.10a.

A byteslice of data from the Memory Array is transferred into the I/O Registers by the operation

* LOAD IO

STORE IO	:	REPEAT (1) 8 TIMES	(1)
	1)	I/O --> MA(D)	(1)
		SHIFT I/O	
STORE IOCTJ	:	REPEAT (1) 8 TIMES	(1)
	1)	I/O --> MA(D)CTJ	(1)
		SHIFT I/O	
LOAD IO	:	REPEAT (1) 8 TIMES	(1)
	1)	MA(S) --> I/O	;S (1)
		SHIFT I/O	



Fig. 5.10a LOAD IO

Each of these operations is executed in 9 clock cycles. One clock

cycle is overhead, caused by setting up the loop in the Microprogram Controller and 8 clock cycles are consumed by shifting 8 bitslices into the I/O Registers.

For illustration, we give in Figure 5.10b the complete microprogram executing the LOAD IO operation. The first microinstruction sets up a loop to be executed 8 times. The loop consists of one microinstruction only, shifting a bitslice from an address released from register 0 in the Address Processor to the I/O Register. Register 0 in the Address Processor is simultaneously incremented.

	LABEL SET 07H
LOAD_IO:	DEF LDCT,RBO
	LABEL SET COPY
	AM SHIFT
COPY:	DEF INCB,RBO,RPCT

Fig. 5.10b LOAD IO

2) Transferring a byte from a selected I/O Register to the I/O Buffer Register.

The contents of the I/O Register of the first memory word with the Tag set is transferred to the I/O Buffer Register by the operation

* LOAD IOB, shown in Figure 5.11.

LOAD IOB : 1) I/O --> IOB	(1)
---------------------------	-----

IOB

X

B

I/O

T

A
B
E

0
1
1

A
B
E

0
1
0

Before

After

Fig. 5.11 LOAD IOB

This operation is executed in 1 clock cycle.

3) Transferring a byte from the I/O Buffer Register to all I/O Registers

The contents of the I/O Buffer Register are spread to all I/O Registers by the operation

* SPREAD IOB, shown in Figure 5.12.

SPREAD IOB : 1) IOB --> I/O.

(1)

IOB

A

A

I/O Comparand

X

A

I/O Mask

X

A

I/O

X
XA
A

Before

After

Fig. 5.12 SPREAD IOB

This operation is executed in 1 clock cycle.

4) Transferring a byte from the I/O Registers to the Comparand and Mask Registers.

A byte is loaded into the Comparand and Mask Registers from their associated I/O Registers by the following operations:

* LOAD COMPARAND, shown in Figure 5.13

* LOAD MASK

LOAD COMPARAND :	REPEAT (1) 8 TIMES	(1)
1) I/O --> COMPARAND(D)		(1)
	SHIFT I/O	
LOAD MASK :	REPEAT (1) 8 TIMES	(1)
1) I/O --> MASK(D)		(1)
	SHIFT I/O	

I/O Comparand D

A

X

A

A

Before

After

Fig. 5.13 LOAD COMPARAND

These operations are executed in 9 clock cycles each.

5) Moving a byteslice.

A byteslice may be transferred from one address to another by the operations:

- * MOVE BYTESLICE, in all words of the Memory Array
- * MOVE BYTESLICE[T], shown in Figure 5.14, only in words where the Tag is set to one.

MOVE BYTESLICE :	REPEAT (1) 8 TIMES	(1)
1) MOVE BITSLICE		(2)
MOVE BYTESLICE[T] :	REPEAT (1) 8 TIMES	(1)
1) MOVE BITSLICE[T]	;S->D	(2)

MA

S

D

T

A	X
B	X

0
1

A	X
B	B

0
1

Before

After

Fig. 5.14 MOVE BYTESLICE[T]

The execution of each of the two operations takes 17 clock cycles.

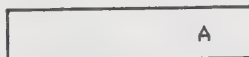
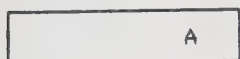
6) Transferring a byte from the Comparand Register to the Memory Array.

Bits from the Comparand Register are successively loaded into all R flip-flops and stored in parallel in the Memory Array by operations:

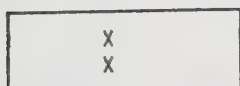
- * STORE CBYTE in all words of the Memory Array
- * STORE CBYTECTJ, shown in Figure 5.15, in the memory words with the Tag set to one.

STORE CBYTE	:	REPEAT (1,2) 8 TIMES	(1)
		1) COMPARAND --> R	(1)
		2) STORE R	(1)
STORE CBYTECTJ	:	REPEAT (1,2) 8 TIMES	(1)
		1) COMPARAND --> R	;S (1)
		2) STORE RCTJ	;D (1)

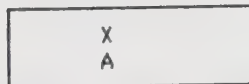
Comparand S



MA D



T



Before

After

Fig. 5.15 STORE CBYTECTJ

The execution of the operations takes 17 clock cycles each.

7) Transferring a byte from a selected memory word to a byteslice in all memory words.

The operation

- * SPREAD BYTE

transfers one byte from an address in a selected memory word to a byteslice at another address in all the memory words. Figure 5.16 illustrates the implementation. The execution proceeds in four steps. In step 1 the byteslice including the byte to be spread, is copied into the I/O Registers in 9 clock cycles. Then, in steps 2 and 3, the selected byte is spread to all I/O Registers in 2 clock cycles. Finally, in step 4, the contents of the I/O Registers are stored into the Memory Array in 8 clock cycles. Step 4 takes only 8 clock cycles instead of 9 because setting up the loop, which takes 1 clock cycle, can overlap with step 3.

SPREAD BYTE	:	1) LOAD IO	;S	(9)
		2) LOAD IOB		(1)
		3) SPREAD IOB		(1)
		4) STORE IO	;D	(8)

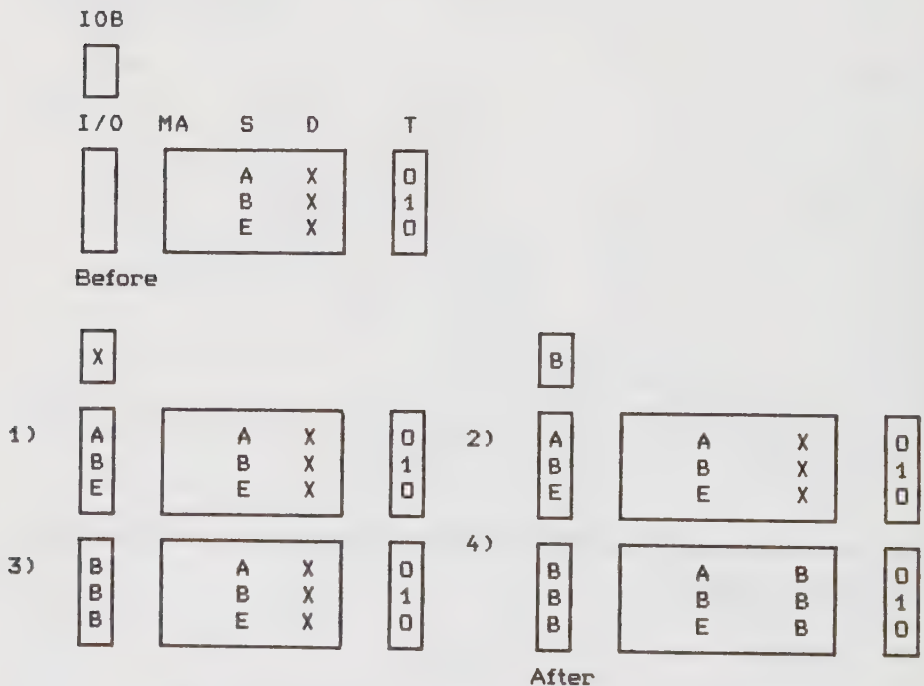


Fig. 5.16 SPREAD BYTE

The execution of the SPREAD BYTE operation takes 19 clock cycles.

8) Transferring a byte from a selected memory word into the Comparand Register.

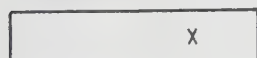
The operation

* LOAD CBYTE

transfers one byte from a selected memory word at one address to the Comparand Register at another address. This operation is illustrated in Figure 5.17.

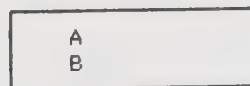
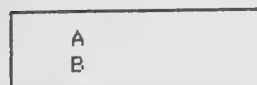
LOAD CBYTE	:	1) LOAD IO	;S	(9)
		2) LOAD IOB		(1)
		3) SPREAD IOB		(1)
		4) LOAD COMPARAND	;D	(8)

Comparand D



MA S

T



Before

After

Fig. 5.17 LOAD CBYTE

The execution of this operation takes 19 clock cycles.

5.2.3 WORD TRANSFER OPERATIONS

Operations in this group transfer words of data between different parts of the Associative Array.

1) Moving word.

The operation

* MOVE WORD, shown in Figure 5.18

transfers one word of b bytes of data from an address in one memory word to another address in all the selected memory words.

MOVE WORD	:	REPEAT (1,2,3,4,5,6)	b TIMES	(1)
	1)	LOAD	IO ;S	(9)
	2)	LOAD	T ;SM	(1)
	3)	LOAD	IOB	(1)
	4)	SPREAD	IOB	(1)
	5)	LOAD	T ;DM	(1)
	6)	STORE	IOCTJ ;D	(8)

MA S D SM DM

XXX	XXX	0	1
ABC	XXX	1	1
EBA	XXX	0	0

XXX	ABC	0	1
ABC	ABC	1	1
EBA	XXX	0	0

Before

After

Fig. 5.18

The execution of this operation takes $22 \cdot b$ clock cycles. One clock cycle is overhead, caused by testing whether all b bytes of the word have been transferred.

Step 6 takes only 8 clock cycles instead of 9 because setting up the loop, which takes 1 clock cycle, can overlap with step 5.

2) Moving a field.

A field of words in the Memory Array is uniquely identified by the size of the words in bytes, by a bitaddress to its rightmost bit and by a Markbitslice indicating in which memory words it is defined. The operation

* MOVE FIELD, shown in Figure 5.19

transfers a field of words from one address in the Memory Array to another one. The Markbitslice is also transferred.

MOVE FIELD: 1) MOVE BITSlice	;Mark bitslice (2)
REPEAT (2) b TIMES	;b bytes (1)
2) MOVE BYTESlice[]	;Byte of word (17)

MA S D SM DM

XXX	XXX	0	x
ABE	XXX	1	x
EBA	XXX	1	x

Before

XXX	XXX	0	0
ABE	ABE	1	1
EBA	EBA	1	1

After

Fig. 5.19 MOVE FIELD

The execution of this operation takes $17*b + 3$ clock cycles.

3) Operation loading a selected word into the Comparand Register.

The operation

* LOAD CWORD, shown in Figure 5.20a

loads a selected word in the Comparand Register.

LOAD CWORD	:	REPEAT (1) b TIMES	(1)
		1) LOAD CBYTE	(19)

Comparand D

XXX

ABE

MA S T

EBA
ABE

0
1

EBA
ABE

Before

After

Fig. 5.20a LOAD CWORD

The execution of this operation takes $20*b$ clock cycles.

For illustration, we give the microprogram for the LOAD CWORD operation in Figure 5.20b. The source address of the word in the Memory Array is in register 0 in the Address Processor, the destination address is in register 1 and the size of the word is in register 2.

```

LOAD_CWORD: LABEL SET 07H
              DEF LDCT,RB0
              LABEL SET COPY1
              AM SHIFT
COPY1:        DEF INCB,RB0,RPCT ;byte to I/O regs
              AM IORS
              DEF LDIO          ;I/O reg -> I/O buff
              LABEL SET 07H
              AM IOWRA
              DEF ENIO,LDCT,RB1 ;I/O buff -> I/O regs
              LABEL SET COPY2
              AM SHIFT
COPY2:        DEF INCB,RB1,WRC,RPCT ;I/O reg -> Comp
              LABEL SET LOAD_CWORD
              DEF CJP,ZAP,DECB,RB2

```

Fig. 5.20b LOAD CWORD

4) Storing a word in the Comparand Register in the Memory Array.

The operation

* STORE CWORD, shown in Figure 5.21

transfers a word in the Comparand Register into the selected memory words.

STORE CWORD	:	REPEAT (1) b TIMES	(1)
		1) STORE CBYTECTJ	(17)

Comparand

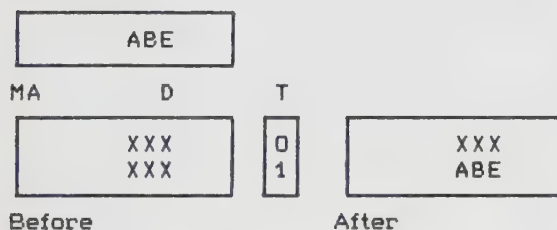


Fig. 5.21 STORE CWORD

The execution of this operation takes $18 \cdot b$ clock cycles.

5.2.4 COMPARE OPERATIONS

Many different types of basic operations for comparing data in the Associative Array can be implemented. We can classify them according to two criteria:

* The location of compared data in the Associative Array.

In one group of operations, a word in the Comparand Register is compared to a field of words in the Memory Array. This is the classical 'one to many' comparison, typical for associative memories, which makes them attractive for designers of computer systems. The execution time for the operation is independent of the number of words in the field. Another group of operations is the one where two fields of data in the Memory Array are compared with each other in parallel. Two data words in each memory word are compared with each other.

* The type of comparison. There are many properties of data that can be used for comparison. The simplest type of comparison is the exact match. In an operation executing the exact match, all corresponding bits are tested for equality in all pairs of words. More complex comparisons are common in cases where the data are interpreted as numbers. Comparisons can then be of the type: greater than, less than etc.

Sometimes, only part of a field of data in the Memory Array is to be interrogated during a compare operation. This feature can be implemented in two ways:

* Using the Mask Register. The content of the Mask Register at the current bitaddress indicates to the Control Unit that the execution of an operation must be disabled in those bitslices.

* Using the Address Processor. The Address Processor,

when generating the sequence of bitaddresses to the field, skips the parts of the field that should not be compared.

For use in algorithms implementing the relational algebraic operations the most important compare operation is

* EXACT MATCH TO COMPARAND, shown in Figure 5.22

In this operation, each bit of a word in the Comparand Register is simultaneously compared to the corresponding bit in all memory words and the Tags are reset to zero if the bits are not equal.

EXACT MATCH TO COMPARAND :	REPEAT b TIMES	(1)
	REPEAT (1) 8 TIMES	(1)
	1) COMPARE 1 BIT	(1)

Comparand

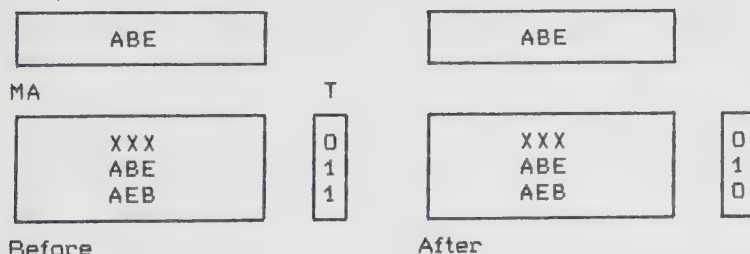


Fig. 5.22a EXACT MATCH TO COMPARAND

The execution time for this operation is $10 \cdot b$ clock cycles.

In Figure 5.22b we give the microprogram for the EXACT MATCH TO COMPARAND operation. The address to the words in the Memory Array is in register 0 of the Address Processor and the size of the word is in the Loop-counter Register.

	LABEL SET 07H
E_M_C:	DEF LDCT,RBO,DECLC
	LABEL SET COMP
	AM CMCT,D
COMP:	DEF INCB,RBO,RPCT
	LABEL SET E_M_C
	DEF CJP,NZLC

Fig. 5.22b EXACT MATCH TO COMPARAND

5.3 INTERNAL ALGORITHMS FOR ALGEBRAIC OPERATIONS

In Section 5.2 we described algorithms for some representative basic operations in the Associative Array. Now we will use these basic operations when describing the implementation of algorithms for algebraic operations. Details of the implementation which are not essential for understanding of the main ideas behind the algorithms will be omitted.

The result of an algebraic operation, taking one or two relations as its arguments, is a new relation. Depending on where the resulting relation is located, relative to the argument relations, the algorithms for operations can be divided into two groups:

- * Algorithms where the result relation is a physical subset of one of the argument relations. The algorithms determine which pieces of data in the Memory Array belong to the result by creating the Markbitslice of the result relation. In this group we find the algorithms for: Selection, Intersection, Difference, Semi-join, Projection and Division.
- * Algorithms which assemble the result relation in some new area in the Memory Array from pieces of data of the argument relations. In this group are the algorithms for: Union, Product and Join.

To simplify our exposition, and to make the timing equations comprehensible, we limit ourselves to relations having only one or two attributes. This can be done without any greater loss of generality.

5.3.1 ALGORITHMS CREATING A MARKBITSlice OF A RESULT

SELECTION

The result of the Selection operation is a relation which is a horizontal subset of data of the argument relation. During execution of the operation a datum in the Comparand Register is compared in parallel with values of an attribute in all tuples of the relation in the Memory Array. The result consists of tuples satisfying the criterion for comparison. As there can be many different conditions for comparison, there can be many different Selection operations. We will show the implementation when the criterion for comparison is equality.

Figure 5.23 illustrates the algorithm for the Selection operation.

The algorithm for the Selection operation has four parameters:

- * The address to the attribute which will be compared with the contents of the Comparand Register (A)
- * The size, in bytes, of the attribute (b)
- * The address to the Workfield of the argument relation (AWF)
- * The address to the Workfield of the result (RWF).

All four parameters are loaded in the Address Processor in the Control Unit before the execution.

The execution proceeds in the following four steps:

First, the Markbitslice of the argument relation is loaded into the Tags in step 1. Then, the Exact Match To Comparand is performed, with the outcome in the Tags, in step 2. Finally, the Tags are stored in the Markbitslice of the result relation in steps 3 and 4.

```
SELECTION: 1) LOAD T      ;argument Markbitslice (1)
           2) EXACT MATCH TO COMPARAND          10*b
           3) T --> R      (1)
           4) STORE R      ;result Markbitslice (1)
```

Comparand

LONDON

	A	AWF	
			RWF
MA	I	I	I
XXXXX	XXXXXX	0	x
SMITH	LONDON	1	x
JONES	PARIS	1	x
BLAKE	PARIS	1	x
CLARK	LONDON	1	x
ADAMS	ATHENS	1	x

XXXXX	XXXXXX	0	0
SMITH	LONDON	1	1
JONES	PARIS	1	0
BLAKE	PARIS	1	0
CLARK	LONDON	1	1
ADAMS	ATHENS	1	0

Before

After

Fig. 5.23 Selection

The execution of this operation takes

$$T_{\text{Selection}} = 10*b + 3 \text{ clock cycles.}$$

b is the size of the attribute.

It is not necessary that the datum in the Comparand Register is at the same physical address as the attribute to which it is compared. The algorithm for the Selection operation has in this case five parameters. Besides the four above, it also needs the address to the datum in the Comparand Register. The execution time for this operation is

$$T_{\text{Selection}} = 18*b + 3 \text{ clock cycles.}$$

The reason why this takes longer than the previous operation is that

for each comparison of a bitslice in the Memory Array with a bit from the Comparand Register, data must be read from two different addresses.

INTERSECTION

The Intersection operation takes two relations as its arguments. The result relation is a horizontal subset of one of the argument relations consisting of tuples which belong to both relations.

The principle behind the implementation of the Intersection is that the result relation is determined by a successive identification of its tuples in its 'mother' relation. The tuples from one relation are transferred into the Comparand Register above the other relation, and compared with it in parallel for an exact match. If the tuple in the Comparand Register is identical with some tuple in the relation below, its original in the first relation is added to the result relation.

Figure 5.24 illustrates the implementation of the Intersection operation.

The Intersection operation has six parameters:

- * The address to the first argument relation (A)
- * The size, in bytes, of the tuple (b)
- * The address to the second argument relation (B)
- * The address to the Workfield of the first relation (AWF)
- * The address to the Workfield of the second relation (BWF)
- * The address to the Workfield of the result relation (RWF).

The execution proceeds in the following 11 steps:

First, the Markbitslice of the result relation is cleared in steps 1 and 2. Then, the Markbitslice of the first argument relation is copied into a scratch pad bitslice in a Workfield of the first relation in steps 3 and 4. The main loop of the algorithm consists of steps 5-11. The loop will be executed once for each tuple of the first argument relation. The next tuple of the first argument relation is selected in step 5. If all the tuples of the first relation have been processed the loop is aborted and the execution of the operation is finished. The information about which memory word holds the selected tuple is saved in R flip-flops in step 6. The selected tuple is transferred into the Comparand register above the second relation in step 7. In steps 8 and 9, the tuple in the Comparand Register is compared with all tuples of the second argument relation. If, after the comparison, all Tags are reset, indicating that the current tuple does not belong to both relations, then, as a result of the test in step 10, the main loop will start again and step 5 will be executed once more. In the other case the position of the current tuple in the first argument relation is added into the Markbitslice of the result relation in step 11.

```

INTERSECTION: 1) CLEAR R          ;0 --> R          (1)
                2) STORE R         ;RWF              (1)
                3) LOAD  R          ;AWF              (1)
                4) STORE R          ;scratch pad      (1)
                   REPEAT (5,6,7,8,9,10,11)
                5) SELECT FIRST AND REMOVE            (5)
                   ;argument1
                6) T --> R          (1)
                7) LOAD WORD TO COMPARAND             20*b
                8) LOAD T          ;BWF              (1)
                9) EXACT MATCH TO COMPARAND           10*b
                   ;argument2
               10) IF NONE GOTO 5          (1)
               11) R OR MA(RWF) --> MA(RWF)          (2)
                   ;update result Markbitslice

```

	A	AWF			
		B	BWF		
MA	R1				RWF
	I	I	I	I	I

LONDON	PARIS	1	1	x
ATHENS	XXXXXX	1	0	x
PARIS	ROME	1	1	x
XXXXXX	LONDON	0	1	x
MADRID	BERLIN	1	1	x
ROME	HAAG	1	1	x

LONDON	PARIS	1	1	1
ATHENS	XXXXXX	1	0	0
PARIS	ROME	1	1	1
XXXXXX	LONDON	0	1	0
MADRID	BERLIN	1	1	0
ROME	HAAG	1	1	1

Before

After

Fig. 5.24 Intersection

The execution time of the Intersection operation is

$$T_{\text{Intersection}} = 7 + (8 + 30 \cdot b) \cdot N_1 + 2 \cdot p \cdot N_1 \text{ clock cycles.}$$

N_1 is the cardinality of the first argument relation, b is the size of the tuple, and p is the ratio between the cardinality of the result and N_1 .

DIFFERENCE

The Difference operation has two relations as its arguments. The result relation is a horizontal subset of one of the argument relations, consisting of tuples which do not belong to the second relation.

The idea behind the implementation is that at the start it is assumed that the result relation is equal to the first argument relation. Then it is successively tested whether the tuples of this relation belong to the second relation or not. If they do they are eliminated from the result.

Figure 5.25 illustrates the implementation of the Difference operation.

The Difference operation has six parameters:

- * The address to the first relation (A)
- * The size, in bytes, of the tuple (b)
- * The address to the second relation (B)
- * The address to the Workfield of the first relation (AWF)
- * The address to the Workfield of the second relation (BWF)
- * The address to the Workfield of the result relation (RWF).

The execution proceeds in the following 10 steps:

First, the Markbitslice of the first argument relation is copied into the Markbitslice of the result relation and also into a scratch pad bitslice in the Workfield of the first relation in steps 1, 2 and 3. The main loop of the algorithm consists of steps 4-10, and it is executed once for each tuple of the first argument relation. The next tuple of the first argument relation is selected in step 4. If all tuples of the first relation have been processed then the

execution is terminated. The information about which memory word holds the selected tuple is saved in R flip-flops in step 5 and the selected tuple is transferred into the Comparand Register above the second relation in step 6. In steps 7 and 8, the tuple in the Comparand Register is compared for identity with all tuples of the second argument relation. If the result of the comparison is negative and no tuple is identical with the tuple in the Comparand Register then the main loop will start again and step 4 will be executed next. Else the position of the current tuple is deleted from the Markbitslice of the result in step 10.

```

DIFFERENCE: 1) LOAD R      ;AWF                      (1)
              2) STORE R   ;RWF                      (1)
              3) STORE R   ;scratch pad              (1)
                REPEAT (4,5,6,7,8,9,10)
              4) SELECT FIRST AND REMOVE             (5)
                  ;argument1
              5) T --> R                               (1)
              6) LOAD WORD TO COMPARAND              20*b
                  ;argument1
              7) LOAD T      ;BWF                    (1)
              8) EXACT MATCH TO COMPARAND            10*b
                  ;argument2
              9) IF NONE GOTO 4                      (1)
             10) R XOR MA(RWF) --> MA(RWF)           (2)
                  ;update result Markbitslice
  
```

	A		B	AWF	BWF	RWF
MA	R					
	I		I	I	I	I

LONDON	PARIS	1	1	x
ATHENS	XXXXXX	1	0	x
PARIS	ROME	1	1	x
XXXXXX	LONDON	0	1	x
MADRID	BERLIN	1	1	x
ROME	HAAG	1	1	x

LONDON	PARIS	1	1	0
ATHENS	XXXXXX	1	0	1
PARIS	ROME	1	1	0
XXXXXX	LONDON	0	1	0
MADRID	BERLIN	1	1	1
ROME	HAAG	1	1	0

Before

After

Fig. 5.25 Difference

The execution time of the Difference operation is

$$T_{\text{Difference}} = 6 + (8 + 30 \cdot b) \cdot N_1 + 2 \cdot p \cdot N_1 \text{ clock cycles.}$$

$N1$ is the cardinality of the first argument relation, b is the size of the tuple, and p is the ratio between the number of tuples from the first relation which also belong to the second relation and $N1$.

SEMI-JOIN

The Semi-join operation has two relations as its arguments. The first argument relation has two attributes, $A1$ and $A2$, and the second has one attribute, B . Values of $A2$ and B are drawn from the same domain. The result relation is a subset of the first argument relation, consisting of tuples where the value of the attribute $A2$ is the same as some value of the attribute B of the second relation. The implementation is similar to the implementation of the Intersection operation in that the tuples of the result are successively identified and added to the result relation.

Figure 5.26 illustrates the implementation of the Semi-join operation.

The Semi-join operation has seven parameters:

- * The address to the first attribute of the first relation ($A1$)
- * The address to the second attribute of the first relation ($A2$)
- * The address to the second relation (B)
- * The size, in bytes, of the tuple of the second relation (the size of the second attribute of the first relation) (b)
- * The address to the Workfield of the first relation (AWF)
- * The address to the Workfield of the second relation (BWF)

* The address to the Workfield of the result relation (RWF)

SEMI-JOIN:	1) CLEAR R	;0 --> R	(1)
	2) STORE R	;RWF	(1)
	3) LOAD R	;BWF	(1)
	4) STORE R	;scratch pad	(1)
	REPEAT (5,6,7,8,9,10)		
	5) SELECT FIRST AND REMOVE		(5)
		;argument 2	
	6) LOAD WORD TO COMPARAND		20*b
		;above A2	
	7) LOAD T	;AWF	(1)
	8) EXACT MATCH TO COMPARAND		10*b
	9) IF NONE GOTO 5		(1)
	0) T OR MA(RWF) --> MA(RWF)		(3)

	A1	A2	AWF		
			B	BWF	
	R1	R2	RWF		
MA					

SMITH	LONDON	XXXXXX	1	0	x
JONES	PARIS	ATHENS	1	1	x
BLAKE	PARIS	PARIS	1	1	x
XXXXX	XXXXXX	BERLIN	0	1	x
CLARK	LONDON	XXXXXX	1	0	x
ADAMS	ATHENS	XXXXXX	1	0	x

SMITH	LONDON	XXXXXX	1	0	0
JONES	PARIS	ATHENS	1	1	1
BLAKE	PARIS	PARIS	1	1	1
XXXXX	XXXXXX	BERLIN	0	1	0
CLARK	LONDON	XXXXXX	1	0	0
ADAMS	ATHENS	XXXXXX	1	0	1

Before

After

Fig. 5.26 Semi-join

The execution proceeds in the following 10 steps:

The Markbitslice of the result is cleared in steps 1 and 2. In steps 3 and 4, the Markbitslice of the second argument relation is copied into a scratch pad bitslice in the Workfield of the second relation. The main loop consists of steps 5-10, and it is executed once for each tuple of the second relation. In step 5, the next tuple of the second relation is selected. In step 6, it is loaded into the Comparand Register above A2 in the first relation and it is compared with A2 in steps 7 and 8. If there is no value of the attribute A2 which is identical to the current value in the Comparand Register, then the loop is restarted, else the status of the Tags is added by

the operation logical OR to the result Markbitslice in step 10.

The execution time of the Semi-join operation is

$$T_{\text{Semi-join}} = 7 + (7 + 30 \cdot b) \cdot N_2 + 3 \cdot p \cdot N_2 \text{ clock cycles.}$$

N_2 is the cardinality of the second relation, b is the size of the tuple of the second relation, and p is the ratio between the number of tuples of the second relation having the same value as some value of the second attribute of the first relation and N_2 .

PROJECTION

The result of the Projection operation is a relation which is both a vertical and a horizontal subset of the argument relation. Producing the vertical subset is simple, the Supporting Processor must just record which attributes of the argument relation belong to the result relation in the directory with information about relations loaded in the Associative Array. Producing the horizontal subset is computationally more difficult. After an attribute is eliminated from the argument relation, the tuples with remaining data must be compared with each other. If there are some identical tuples, only one must be chosen to belong to the result relation.

There are two variants of the Projection operation, called Projection1 and Projection2, both illustrated in 5.27.

Projection1: If a key attribute in the argument relation, uniquely identifying each tuple, follows after the Projection over into the result relation, then there is no need for checking the redundancy in the result. The existence of the key guarantees that there are no identical tuples in the relation. The execution of Projection1 in the Associative Array consists of copying the Markbitslice of the argument relation into the Markbitslice of the result relation. The algorithm for Projection1 has two parameters:

- * The address to the Markbitslice of the argument (AWF)

- * The address of the Markbitslice of the result (R1WF).

Projection2: If none of the attributes of the argument relation going over to the result relation (or their combination) is a key, the candidates for tuples of the result must be checked for redundancy. The idea behind the implementation is that the nonredundant tuples of the argument relation are successively selected and added into the result relation.

Projection2 has four parameters:

- * The address to the attribute over which it is projected (A2)
- * The size, in bytes, of this attribute (b)
- * The address to the Workfield of the argument relation (AWF)
- * The address to the Workfield of the result (R2WF)

The execution proceeds in the following 11 steps.

The Markbitslice of the result relation is cleared in steps 1 and 2. The Markbitslice of the argument relation is copied into a scratch pad bitslice in a Workfield of the argument relation, in steps 3 and 4, to be used in successive selection of tuples. The main loop of the algorithm consists of steps 5-11, and it is executed once for each tuple of the result. The next tuple of the argument relation is selected in step 5 and transferred into the Comparand Register in step 6. The exact match operation between this tuple and the argument relation is performed in steps 7 and 8. In step 9, all tuples identical to the tuple currently in the Comparand Register are removed from the scratch pad bitslice. The first memory word with such a tuple is selected in step 10, and added into the Markbitslice of the result relation in step 11 by the operation logical OR.

PROJECTION1:	1)	LOAD R	;AWF	(1)
	2)	STORE R	;R1WF	(1)
PROJECTION2:	1)	CLEAR R	;0 --> R	(1)
	2)	STORE R	;R2WF	(1)
	3)	LOAD R	;AWF	(1)
	4)	STORE R	;scratch pad	(1)
		REPEAT (5,6,7,8,9,10,11)		
	5)	SELECT FIRST AND REMOVE		(5)
	6)	LOAD WORD TO COMPARAND		20*b
	7)	LOAD T	;AWF	(1)
	8)	EXACT MATCH TO COMPARAND		10*b
	9)	T OR MA(SP) --> MA(SP)		(2)
	10)	SELECT FIRST		(1)
	11)	T OR MA(R2WF) --> MA(R2WF)		(2)
		;update result Markbitslice		

	A1		A2	AWF	
	R1				R1WF
			R2		R2WF
MA	I	I	I	I	I

XXXXX	XXXXXX	0	x	x
SMITH	LONDON	1	x	x
JONES	PARIS	1	x	x
BLAKE	PARIS	1	x	x
CLARK	LONDON	1	x	x
ADAMS	ATHENS	1	x	x

XXXXX	XXXXXX	0	0	0
SMITH	LONDON	1	1	1
JONES	PARIS	1	1	1
BLAKE	PARIS	1	1	0
CLARK	LONDON	1	1	0
ADAMS	ATHENS	1	1	1

Before

After

Fig. 5.27 Projection

The execution time of the Projection1 operation is

$$T_{\text{Projection1}} = 2 \text{ clock cycles}$$

and the execution of the Projection2 takes

$$T_{\text{Projection2}} = 7 + (11 + 30*b)*NR2 \text{ clock cycles.}$$

b is the size of the attribute on which the argument relation is projected and NR2 is the cardinality of the result relation.

DIVISION

A result of the Division operation is a horizontal and vertical subset of the dividend relation. We assume that the dividend has two attributes, A1 and A2, and that the divisor has one, B. Values of A2 and B must be drawn from the same domain. The implementation of the Division operation proceeds in two phases. In the first phase, tuples from the divisor are successively compared to the attribute A2 of the dividend and a field with a partial result consisting of bitslices with the results of each comparison is created. In the second phase, the information from the partial result field is used in identifying the values of the attribute A1 of the dividend that will become the tuples of the result relation.

Figures 5.28a and 5.28b illustrate the implementation of the Division operation. We give the state of the Memory Array before the computation, after its first phase, and when it is completed.

The Division has eight parameters:

- * The address to the first attribute of the dividend (A1)
- * The address to the second attribute of the dividend (A2)
- * The size, in bytes, of the first attribute (b1)
- * The size, in bytes, of the second attribute (b2)
- * The address to the Workfield of the dividend (AWF)
- * The address to the divisor (B)
- * The address to the Workfield of the divisor (BWF)
- * The address to the Workfield of the result (RWF).

DIVISION:	1)	CLEAR R	;0 --> R	(1)
	2)	STORE R	;RWF	(1)
	3)	LOAD R	;BWF	(1)
	4)	STORE R	;first scratch pad	(1)
		REPEAT (5,6,7,8,9,10)		
PHASE1:	5)	SELECT FIRST AND REMOVE		(5)
		;divisor		
	6)	LOAD WORD TO COMPARAND	20*b2	
		;above A2 in dividend		
	7)	LOAD T	;AWF	(1)
	8)	EXACT MATCH TO COMPARAND	10*b2	
	9)	T --> R		(1)
	10)	STORE R	;partial result array	(1)
	11)	SAVE NB IN THE ADDRESS PROCESSOR		(1)
PHASE2:	12)	LOAD R	;AWF	(1)
	13)	STORE R	;second scratch pad	(1)
		REPEAT (14,15,16,17,18,19,20,21,		
		22,23,24,25,26,27)		
	14)	LOAD R	;second scratch pad	(1)
	15)	SELECT FIRST		(1)
	16)	EXIT IF NONE	;all tuples tested	(1)
	17)	LOAD WORD TO COMPARAND	20*b1	
		;above A1 in dividend		
	18)	LOAD T	;AWF	(1)
	19)	EXACT MATCH TO COMPARAND	10*b1	
	20)	T XOR MA(SP) --> MA(SP)	;scratch pad	(3)
	21)	T --> R	;save T	(1)
		REPEAT (22,23,24) NB TIMES		(2)
	22)	R --> T	;get T	(1)
	23)	MA AND T --> T		(2)
		;bitslice in partial result field and T		
	24)	IF NONE GOTO 14		(1)
		;test of partial result field aborted		
	25)	R --> T	;get T	(1)
	26)	SELECT FIRST		(1)
	27)	R OR MA(RWF) --> MA(RWF)		(2)
		;update result Markbitslice		

Fig. 5.28a Division

The execution proceeds in the following 27 steps:

In steps 1 and 2, the result Markbitslice is cleared. In steps 3 and 4, the Markbitslice of the divisor is copied into a scratch pad bitslice in its Workfield. Phase 1, of the computation proper, consists of steps 5-10, executed once for each tuple of the divisor,

and step 11. The tuples of the divisor are successively transferred to the Comparand Register and compared with the attribute A2 of the dividend in steps 5-8. The result of each comparison is stored in the partial result field in the Memory Array in steps 9 and 10. After Phase 1, a field of bitslices is created in the Memory Array, one for each tuple of the divisor. The information in the n-th bitslice indicate which tuples of the dividend have the same value of the attribute A2 as the n-th tuple of the divisor. During the execution of Phase 1, the Address Processor computes the number (NB) of bitslices in the partial result array and saves this number in one of its internal registers in step 11. This number determines the maximum number of times an inner loop in Phase 2 will be executed. The main loop of Phase 1 is executed NB times.

In Figure 5.28b we can see the array consisting of two bitslices, (I) with the result of the comparison of the first tuple of the divisor and (II) with the result of the comparison of the second tuple of the divisor with the attribute A2 of the dividend.

Phase 2 starts by copying the Markbitslice of the divisor to the scratch pad bitslice in its Workfield, in steps 12 and 13. This scratch pad bitslice is used for successive selection of values of the attribute A1 of the dividend performed in steps 14 and 15. If all different values of the attribute A1 are already tested in the course of the computation, the execution is finished in step 16. In steps 17,18 and 19, the new selected value of the attribute A1 is transferred into the Comparand Register above A1 and compared with it. Step 20 removes from the scratch pad area the information about tuples holding the current attribute value of the attribute A1. The result of the comparison conducted in step 19, the bitslice indicating which Memory Words hold the current attribute value, is saved in the R flip-flops in step 21. In steps 22 and 23, this bitslice is successively compared by the operation logical AND with the bitslices from the partial result field which was created in Phase 1. If for each bitslice of the field the AND operation leaves some Tag set to one, then the information about the first Memory Word with the current attribute is selected in step 26 and added into the result

Markbitslice in step 27 and the main loop of Phase 2 starts again.

The number of times the main loop of Phase 2 is executed is equal to the number of different values of attribute A1 in the dividend relation. The loop comprising steps 22-24 is repeated NB times at most.

MA	A1 A2		B	AWF		BWF	RWF
	R1			I	I	I	I
LONDON	P1		XX	1	0	x	
ATHENS	P3		P1	1	1	x	
PARIS	P2		P2	1	1	x	
XXXXXX	XX		XX	0	0	x	
LONDON	P2		XX	1	0	x	
PARIS	P1		XX	1	0	x	
BERLIN	P2		XX	1	0	x	

Before

		I	II				
		I	I				
LONDON	P1	1	0	XX	1	0	0
ATHENS	P3	0	0	P1	1	1	0
PARIS	P2	0	1	P2	1	1	0
XXXXXX	XX	0	0	XX	0	0	0
LONDON	P2	0	1	XX	1	0	0
PARIS	P1	1	0	XX	1	0	0
BERLIN	P2	0	1	XX	1	0	0

After Phase 1

LONDON	P1		XX	1	0	1
ATHENS	P3		P1	1	1	0
PARIS	P2		P2	1	1	1
XXXXXX	XX		XX	0	0	0
LONDON	P2		XX	1	0	0
PARIS	P1		XX	1	0	0
BERLIN	P2		XX	1	0	0

After

Fig. 5.28b Division

The execution time of the Division operation is

$$T_{\text{Division}} = 13 + (8 + 30 \cdot b_2) \cdot N_2 + (8 + 30 \cdot b_1) \cdot N_{A1} + \\ + ((4 \cdot N_2 + 4) \cdot p + 4 \cdot N_2 \cdot r \cdot (1 - p)) \cdot N_{A1} \text{ clock cycles.}$$

b1 is the size of the first attribute of the dividend, b2 is the size of the second attribute of the dividend, N2 is the cardinality of the divisor, NA1 is the number of different values of the first attribute of the dividend, p is the ratio between the cardinality of the result and NA1, and r is the ratio between the average number of times that the loop (22,23,24) is executed before it is aborted and the maximum number of its executions (N2).

5.3.2 ALGORITHMS THAT ASSEMBLE THE RESULT

UNION

The Union of two argument relations gives as a result a relation assembled from all tuples of the first relation and those tuples of the second relation which are not already in the first one.

Figure 5.29 illustrates the implementation of the Union operation.

The Union operation has six parameters:

- * The address to the first relation (A)
- * The size, in bytes, of the tuple (b)
- * The address to the second relation (B)
- * The address to the Workfield of the first relation (AWF)
- * The address to the Workfield of the second relation (BWF)
- * The address to the Workfield of the result relation (RWF)

UNION:	1)	LOAD	R	;AWF	(1)
	2)	STORE	R	;RWF	(1)
	3)	LOAD	R	;BWF	(1)
	4)	STORE	R	;scratch pad	(1)
				REPEAT (5,6,7,8,9,10,11,12,14)	
	5)	SELECT FIRST	AND REMOVE		(5)
				;argument2	
	6)	LOAD WORD TO	COMPARAND		20*b
				;above argument	
	7)	LOAD	T	;RWF	(1)
	8)	EXACT MATCH TO	COMPARAND		10*b
	9)	IF SOME THEN	GOTO 5		(1)
	10)	LOAD	T	;RWF	(1)
	11)	T/ -->	T	;invert T	(1)
	12)	SELECT FIRST			(1)
				;get first empty Memory Word	
	13)	STORE WORD FROM	COMPARAND		18*b
	14)	T OR MA(RWF)	--> MA(RWF)		(3)

	A			AWF	
			B		BWF
	R				RWF
MA	I	I	I	I	I

LONDON	PARIS	1	1	x
ATHENS	XXXXXX	1	0	x
PARIS	ROME	1	1	x
XXXXXX	LONDON	0	1	x
MADRID	BERLIN	1	1	x
ROME	HAAG	1	1	x
XXXXXX	XXXXXX	0	0	x
XXXXXX	XXXXXX	0	0	x

LONDON	PARIS	1	1	1
ATHENS	XXXXXX	1	0	1
PARIS	ROME	1	1	1
BERLIN	LONDON	0	1	1
MADRID	BERLIN	1	1	1
ROME	HAAG	1	1	1
HAAG	XXXXXX	0	0	1
XXXXXX	XXXXXX	0	0	0

Before

After

Fig. 5.29 Union

The execution proceeds in the following 14 steps:

The result Markbitslice is created in steps 1 and 2. All tuples of the first relation belong to the result relation. In steps 3 and 4, the Markbitslice of the second argument relation is copied into a scratch pad bitslice in its Workfield. The main loop of the algorithm consists of steps 5-14, and is executed once for each tuple of the second relation. The next tuple of the second relation is selected in step 5, transferred into the Comparand Register above the first

relation in step 6, and compared to it in steps 7 and 8. If this tuple exists in the first relation then the main loop starts again. In steps 10-12, the next empty Memory Word is selected, and in step 13, the tuple from the Comparand Register is transferred to it. In step 14, the Markbitslice of the result is updated.

The execution time for the Union operation is

$$T_{\text{Union}} = 7 + (7 + 30*b)*N2 + (6 + 18*b)*p*N2 \text{ clock cycles.}$$

b is the size of the tuple, N2 is the cardinality of the second argument relation, and p is the ratio between the number of tuples of the second relation added to the result relation and N2.

PRODUCT

The Product operation has two relations as its arguments. The result is the concatenation of all combinations of tuples of the argument relations.

Figure 5.30 illustrates the implementation of the Product operation.

The Product operation has nine parameters:

- * The address to the first relation (A)
- * The address to the second relation (B)
- * The address to the left part of the result relation (R1)
- * The address to the right part of the result relation (R2)
- * The size, in bytes, of the tuple of the first relation (b1)
- * The size, in bytes, of the tuple of the second relation (b2)
- * The address to the Workfield of the first relation (AWF)

* The address to the Workfield of the second relation (BWF)

* The address to the Workfield of the result (RWF)

The execution proceeds in the following 16 steps:

In steps 1 and 2, all bits of the result Markbitslice are set to one. During execution, the result Markbitslice indicates by one which Memory Words are free to receive result tuples. In steps 3 and 4, the Markbitslice of the first argument relation is copied into a scratch pad bitslice in its Workfield. The main loop of the operation consists of steps 5-13 and it is executed once for each tuple of the first argument relation. In steps 5 and 6, the next tuple of the first argument relation is selected and transferred to the proper address in all Memory Words which are not yet occupied by tuples of the result relation. It will form the left part of the result tuples. In steps 7 and 8, the Markbitslice of the second argument relation is copied into a scratch pad bitslice in its Workfield. The loop consisting of steps 9-13 is executed once for each tuple of the second argument relation. In steps 9 and 10 the next tuple of the second argument relation is selected and transferred to the proper address in all Memory Words which are not yet occupied by the result relation. In steps 11-13, the first occurrence of a new concatenated tuple is marked in the Markbitslice of the result. After all tuples of the first relation have been concatenated with all tuples of the second relation, the Markbitslice of the result is created in steps 14-16.

```

PRODUCT:1) SET      R      ;1 --> R      (1)
          2) STORE R      ;RWF            (1)
          3) LOAD  R      ;AWF            (1)
          4) STORE R      ;scratch pad1    (1)
             REPEAT (5,6,7,8,9,10,11,12,13)
          5) SELECT FIRST AND REMOVE ;scratch pad1 (5)
          6) MOVE WORD  ;A -> R1          22*b1
          7) LOAD  R      ;BWF            (1)
          8) STORE R      ;scratch pad2    (1)
             REPEAT (9,10,11,12,13)
          9) SELECT FIRST AND REMOVE ;scratch pad2 (5)
         10) MOVE WORD  ;B -> R2          22*b2
         11) LOAD  T      ;RWF            (1)
         12) SELECT FIRST                  (1)
         13) T XOR MA(RWF) --> MA(RWF)    (2)
         14) LOAD  T      ;RWF            (1)
         15) T/ --> R      (1)
         16) STORE  R      ;RWF            (1)

```

A		B		AWF			
				R1		BWF	
				R2		RWF	
MA	I	I		I	I	I	I
XXXXXX	XXXXXX	XXXXXX	XXXXXX	0	0	x	
XXXXXX	LONDON	XXXXXX	XXXXXX	0	1	x	
SMITH	XXXXXX	XXXXXX	XXXXXX	1	0	x	
CLARK	PARIS	XXXXXX	XXXXXX	1	1	x	
BLAKE	XXXXXX	XXXXXX	XXXXXX	1	0	x	
XXXXXX	XXXXXX	XXXXXX	XXXXXX	0	0	x	

Before

XXXXXX	XXXXXX	SMITH	LONDON	0	0	1
XXXXXX	LONDON	SMITH	PARIS	0	1	1
SMITH	XXXXXX	CLARK	LONDON	1	0	1
CLARK	PARIS	CLARK	PARIS	1	1	1
BLAKE	XXXXXX	BLAKE	LONDON	1	0	1
XXXXXX	XXXXXX	BLAKE	PARIS	0	0	1

After

Fig. 5.30 Product

The execution time of the Product operation is

$$T_{\text{Projection}} = 10 + (10 + 22*b1 + (9 + 22*b2)*N2)*N1 \text{ clock cycles.}$$

b1 is the size of the tuple of the first relation, b2 is the size of the tuple of the second relation, N1 is the cardinality of the first

and N2 of the second relation.

JOIN

Among different variants of the Join operation we will demonstrate Join2, where the result, obtained by concatenating tuples of two argument relations satisfying some specified condition, does not contain the joining attributes.

We assume that both the first and the second argument relation have two attributes called A1 and A2, and B1 and B2, respectively. The joining attributes are A2 and B2, the condition for joining is their equality. The idea behind the implementation is that values from the attribute A2 of the first relation are successively transferred to the Comparand Register and compared with A2 and B2. Thus selected tuples from the first and the second relation are subsequently concatenated by the Product operation into the tuples of the result relation.

Figure 5.31 illustrates the execution.

The Join2 operation has eleven parameters:

- * The address to the first attribute of the first relation (A1)
- * The size, in bytes, of the first attribute (b1)
- * The address to the second attribute of the first relation (A2)
- * The size, in bytes, of the second attribute (b2)
- * The address to the first attribute of the second relation (B1)
- * The size, in bytes, of the first attribute of the second relation (b3)

- * The address to the second attribute of the second relation (B1)
- * The address to the first attribute of the result relation (R1)
- * The address to the second attribute of the result relation (R2)
- * The address to the Workfield of the first relation (AWF)
- * The address to the Workfield of the second relation (BWF)
- * The address to the Workfield of the result relation (RWF)

The execution proceeds in the following 20 steps:

In steps 1 and 2 all bits of the Markbitslice of the result are set to one. In steps 3 and 4, the Markbitslice of the first argument relation is copied into a scratch pad bitslice in its Workfield. The main loop of the execution consists of steps 5-20, and is executed once for each different value of the attribute A2. In steps 6-9, the next selected value of A2 is transferred into the Comparand Register above A2 and B2. In steps 10 and 11, this value is compared with A2, and in steps 12 and 13, the bitslice indicating selected tuples of the first relation is saved in the second scratch pad bitslice. In step 14, all selected tuples are removed from the list of remaining tuples of the first relation. In steps 15 and 16, the current value in the Comparand Register is compared with the attribute B2 of the second relation. In step 17, the result of the comparison is tested and if no value of the attribute B2 is identical to the current value, the loop starts again. In steps 18 and 19, the result of the comparison is saved in the third scratch pad bitslice. At this point of the execution, the second scratch pad bitslice contains the information about the tuples of the first relation and the third scratch pad bitslice holds the information about the tuples of the first relation which have the same value of the attributes A2 and B2.

These two sets of tuples are combined in step 20 by a modified Product operation.

The execution time of the Join2 operation is

$$\begin{aligned}
 T_{\text{Join2}} = & 7 + (11 + 48*b2)*p*NA2 + \\
 & + (13 + 48*b2)*(1 - p)*NA2 + \\
 & + T_p*NA2*(1-p) \quad \text{clock cycles.}
 \end{aligned}$$

b2 is the size of the join attribute, NA2 is the number of different values in the join attribute of the first relation, p is the number of values of the join attribute of the first relation which do not match any value in the second relation, T_p is the average time for the Product operation.


```

JOIN2:1) SET      R          ;1 --> R          (1)
      2) STORE R          ;RWF                (1)
      3) LOAD  R          ;AWF                (1)
      4) STORE R          ;scratch pad1        (1)
          REPEAT (5,6,7,8,9,10,11,12,13,14,15,16
                                17,18,19 20)
      5) LOAD  T          ;scratch pad1        (1)
      6) SELECT FIRST      (1)
      7) EXIT IF NONE      (1)
      8) LOAD WORD TO COMPARAND ;above A2
      9) LOAD WORD TO COMPARAND ;above B2      28*b1
     10) LOAD  T          ;scratch pad1        (1)
     11) EXACT MATCH TO COMPARAND ;A2          10*b1
     12) T --> R          (1)
     13) STORE R          ;scratch pad2        (1)
     14) T XOR MA(SP1) --> MA(SP1)            (3)
     15) LOAD  T          ;argument2 Markbitslice (1)
     16) EXACT MATCH TO COMPARAND ;B2          10*b1
     17) IF NONE GOTO 5    (1)
     18) T --> R          (1)
     19) STORE R          ;scratch pad3        (1)
     20) PRODUCT          Tpl

```

A1		A2		AWF						
				B1	B2			BWF		RWF
MA	I	I		I	I	R1	R2	I	I	I
LONDON	P1		PARIS	P1	XXXXXX	XXXXXX		1	1	x
XXXXXX	XX		OSLO	P1	XXXXXX	XXXXXX		0	1	x
ATHENS	P1		PARIS	P2	XXXXXX	XXXXXX		1	1	x
BERLIN	P2		XXXXXX	XX	XXXXXX	XXXXXX		1	0	x
XXXXXX	XX		XXXXXX	XX	XXXXXX	XXXXXX		0	0	x
XXXXXX	XX		XXXXXX	XX	XXXXXX	XXXXXX		0	0	x

Before

LONDON	P1	PARIS	P1	LONDON	PARIS	1	1	1
XXXXXX	XX	OSLO	P1	LONDON	OSLO	0	1	1
ATHENS	P1	PARIS	P2	ATHENS	PARIS	1	1	1
BERLIN	P2	XXXXXX	XX	ATHENS	OSLO	1	0	1
XXXXXX	XX	XXXXXX	XX	BERLIN	PARIS	0	0	1
XXXXXX	XX	XXXXXX	XX	XXXXXX	XXXXXX	0	0	0

After

Fig. 5.31 Join2

5.4 DISCUSSION

One of the fundamental results of Computer Science is the insight that any hardware structure (with some necessary minimal capabilities) can, in principle, compute any computable function. Consequently, the sole fact that operations of relational algebra can be implemented on LUCAS is hardly surprising. Rather more interesting is the question whether they can be implemented efficiently or not.

The value of the material presented in the previous section lies in the fact that we could use the rather tedious demonstration of algorithms for deriving exact timing equations. The Associative Array of LUCAS may serve as a model of a special hardware component of a database computer, larger than LUCAS but with similar properties, and the timing equations can be used in a quantitative analysis of the feasibility of using this component.

In this section, we will examine the performance of the Associative Array.

- 1) We will determine the execution time of algebraic operations for typical sizes of the Associative Array and for typical values of parameters.
- 2) We will also compare the execution times on the Associative Array with the execution times on an ordinary sequential computer.
- 3) Finally, we will compare the performance of the Associative Array of LUCAS with some other similar designs.

5.4.1 TIMING EQUATIONS

The analysis of the timing equations is somewhat complicated by the fact that the execution time is dependent not only on the size of argument relations but also on their contents. For example, the execution time of the Projection operation is proportional to the

cardinality of the result relation. In order to be able to determine the very important average behaviour of the algorithms, it is in some cases necessary to know the statistical properties of a database, e.g. the expected number of matching tuples.

We assume a clock frequency of 5 MHz. The size of the Associative Array will vary from 128 processors (the size of LUCAS today) to 32k processors (the size of the Associative Array which we believe can be built in the future).

Table 5.1 summarizes the timing equations which give the number of clock cycles for each operation. The definitions of parameters can be found in Section 5.3.

Selection	$10*b + 3$
Intersection	$7 + (8 + 30*b)*N1 + 2*p*N1$
Difference	$6 + (8 + 30*b)*N1 + 2*p*N1$
Semi-join	$7 + (7 + 30*b)*N2 + 3*p*N2$
Projection1	2
Projection2	$7 + (11 + 30*b)*NR2$
Division	$13 + (8 + 30*b2)*N2 + (8 + 30*b1)*NA1 + ((4*N2 + 4)*p + 4*N2*r*(1-p))*NA1$
Union	$7 + (7 + 30*b)*N2 + (6 + 18*b)*p*N2$
Product	$10 + (10 + 22*b1 + (9 + 22*b2)*N2)*N1$
Join2	$(11 + 48*b2)*p*NA2 + (13 + 48*b1)*(1-p)*NA2 + TP*NA2*(1-p)$

Tab. 5.1 Execution times on LUCAS

The size of the Associative Array is not a parameter in the timing equations. This means that the execution time for an operation on

data loaded in an Associative Array with a given size is the same as the execution time on any larger Array. Hence, the relations will always be assumed to be of the largest possible cardinality for the given size of the Associative Array.

The timing equations in Table 5.1 are too complex to be apprehensible and must be approximated. We assume that the size of tuples, b , and the sizes of attributes, b_1 and b_2 , are larger than 10 bytes. Furthermore, we assume that b_1 is equal to b_2 and, since b_1 and b_2 do not appear in the same equations as b , we call them b , for the reason of uniformity. The parameters p and r , which reflect the statistical properties of the database, have values between 0 and 1.

Based on these assumptions we get the approximate timing equations summarized in Table 5.2.

Selection	$10 \cdot b$
Intersection	$30 \cdot b \cdot N_1$
Difference	$30 \cdot b \cdot N_1$
Semi-join	$30 \cdot b \cdot N_2$
Projection1	2
Projection2	$30 \cdot b \cdot N_2$
Division	$30 \cdot b \cdot (N/N_2 + N_2) + 4 \cdot N$
Union	$30 \cdot b \cdot N_2 + 18 \cdot b \cdot p \cdot N_2$
Product	$22 \cdot b \cdot N_1 \cdot N_2$
Join2	$48 \cdot b \cdot N_2 + N_2 \cdot (1-p) \cdot TP$

Tab. 5.2 Approximate execution times on LUCAS

The approximation of the majority of operations is straightforward. Only the approximation of the Division operation deserves some

comment:

After eliminating the initialization and the termination overhead we get

$$T_{\text{Division}} = 30 \cdot b \cdot N_2 + 30 \cdot b \cdot NA_1 + 4 \cdot N_2 \cdot p \cdot NA_1 + 4 \cdot N_2 \cdot r \cdot (1-p) \cdot NA_1$$

This formula is still too complex. We must analyze its constituents to see if it can be further approximated.

The first term, $30 \cdot b \cdot N_2$, is the time spent on comparing N_2 tuples of the divisor with the second attribute of the dividend.

The second term, $30 \cdot b \cdot NA_1$, is the time spent on selecting a next tuple of the dividend for testing whether it belongs to the result or not.

The third term, $4 \cdot N_2 \cdot p \cdot NA_1$, can be divided into two factors, $p \cdot NA_1$, and $4 \cdot N_2$. The first factor is the number of tuples of the result. The second factor is the time it takes to test each such tuple, and it consists of N_2 elementary 4-clock cycle tests.

The fourth term, $4 \cdot N_2 \cdot r \cdot (1-p) \cdot NA_1$, can also be divided into two factors, $(1-p) \cdot NA_1$ and $4 \cdot N_2 \cdot r$. The first factor is the number of tuples of the dividend that failed the test whether they belonged to the result or not. The second factor contains $N_2 \cdot r$ which is the average number of elementary tests giving a positive result before an elementary test with a negative outcome decides that the test sequence is aborted and that the current tuple will not belong to the result.

The sum of the third and the fourth terms in T_{Division} can be at most equal to $4 \cdot NA_1 \cdot N_2$. We substitute and obtain

$$T_{\text{Division}} = 30 \cdot b \cdot N_2 + 30 \cdot b \cdot NA_1 + 4 \cdot N_2 \cdot NA_1$$

The parameter p was lost in the substitution, and the formula can be applied only in certain typical situations. One situation when it cannot be applied is when the dividend and the divisor have the largest possible cardinality, N , and at the same time, NA_1 is also

equal to N . In this case, (and as a matter of fact, in all cases with $N_2=N$ and NA_1 larger than 1), p is equal to 0 and the result of the Division is an empty relation. This is due to a simple combinatorial reason: there are not enough tuples of the dividend to match the divisor. The execution time of this 'pathological' case is $60*b*N + 4*N$ instead of $60*b*N + 4*N^2$ as predicted by the approximate formula.

A more realistic situation is the case when N_2 and NA_1 satisfy the following condition: $NA_1*N_2=N$. It means that one tuple from each of the NA_1 groups of tuples of the dividend theoretically has a chance to be a part of the result. Thus, the formula

$$T_{\text{Division}} = 30*b(N/N_2 + N_2) + 4*N$$

expresses the approximate execution time.

5.4.2 PERFORMANCE EVALUATION

We will compare the performance of the Associative Array with the performance of a conventional sequential computer. To make this comparison meaningful it is very important to carefully determine what performance measure is to be used. The problem is that we will compare the execution time of a well defined system with the performance of a computer about which only one general property, its sequential mode of operation, is assumed.

We will base our comparison method on the following observation: The implementation of algebraic operations on a sequential computer is typically based on sorting and merging of tuples of operand relations [DeWitt82]. For example the Join consists of sorting the operands and merging the sorted relations. Sorting and merging methods are essentially based on comparisons [Knuth73]. Thus, we assume that the primitive operation of a sequential evaluation of algebraic operations is the comparison of two tuples.

We will estimate the number of necessary comparisons for an operation, assuming a 'good' sequential algorithm. (By 'good' we mean an algorithm using a minimum number of comparisons, even though we are aware that this is not the only way of measuring the effectiveness of sorting and merging.) Then, if we divide the (known) execution time for an operation in the Associative Array by this number and by the number of bytes in a tuple (to get rid of different word-lengths) we obtain a measure which we call an Equivalent Sequential Compare time, ESC. (The influence of the length of a tuple in defining ESC could be eliminated because all execution times, cf. Table 5.2, are linear in b.)

ESC is a rather crude measure, but it is useful for drawing some general conclusions. For example, given an Associative Array, if we know the value of ESC for some algebraic operation then we can conclude that e.g. a 16 bit computer must be able to perform fetch and comparison on two 16 bit words in a time less than twice this value, a 32 bit computer in less than four times this value, etc, in order to be able to achieve the same total execution time for this operation as the Array.

Also, we must be aware that ESC will be quite unfavourable to the Associative Array. ESC reflects only comparisons in the CPU, but sorting or merging involve time consuming data movements, housekeeping operations, etc, as well.

Table 5.3 summarizes the number of necessary comparisons in sequential implementation of algebraic operations. We are using results from Knuth, who shows that to sort n elements at most $(n \log n)$ comparisons are required, to sort n elements having m different values $(n \log m)$ comparisons are required, and that two sorted tables with n elements each can be merged using at most $2 \cdot n$ comparisons.

Selection	N
Intersection	$2*N*\log N + 2*N$
	$2 \times \text{sort} + \text{merge}$
Difference	$2*N*\log N + 2*N$
	$2 \times \text{sort} + \text{merge}$
Semi-join	$2*N*\log N + 2*N$
	$2 \times \text{sort} + \text{merge}$
Projection2	$N*\log N^2$
	sort
Division	$N^2*\log N^2 + 2*N*\log N + 2*N$
	$3 \times \text{sort} + \text{merge}$
Union	$N*\log(N/2) + N$
	$2 \times \text{sort (of } N/2) + \text{merge}$
Product	$22*b*N1*N2$
Join2	$2*N1*\log N1 + 2*N1$
	$2 \times \text{sort} + \text{merge}$

Tab. 5.3 Number of comparisons in a sequential implementation

SELECTION

Table 5.4 gives the execution time, in microseconds, for an attribute size of 16 and 64 bytes. It also gives ESC, in nanoseconds.

b / N	128	1 k	8 k	32 k
16	32	32	32	32
64	128	128	128	128
ESC	15	1.9	0.24	0.061

Tab. 5.4 Execution times for Selection

No matter how small or how large a relation is, as long as it fits in the Associative Array the execution time of the Selection operation is the same.

In the case of the Selection operation, where there is no need for sorting or merging, ESC can be interpreted as the average necessary time for comparison of two bytes in order to execute the Selection as fast as the Associative Array. This holds for any device and not only for a sequential computer. (We assume no use of indexing, hashing etc.) If we examine Table 5.4, by using a simple 'back-of-the-envelope' calculation we can prove that even for an Associative Array as large as one with $N = 1024$ processors, it is not impossible to design a device which uses parallel comparators which is faster.

INTERSECTION, DIFFERENCE, SEMI-JOIN

The timing equations for the Intersection, Difference and Semi-join are identical (cf. Table 5.2). For the Intersection and Difference operations, the parameter b is the size of a tuple, and for the Semi-join it is the size of a joining attribute.

Figure 5.32 shows the execution time for two typical sizes (16 and 64) of b .

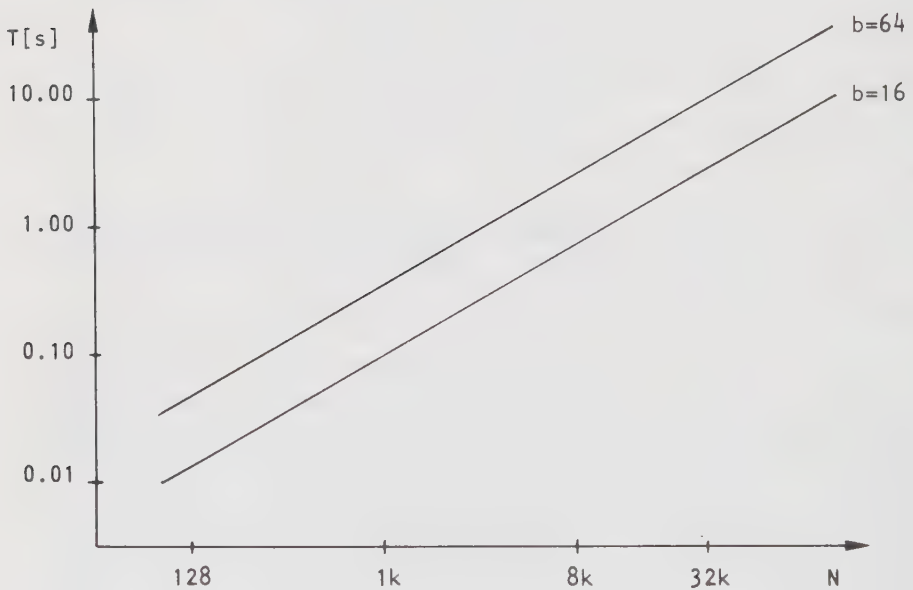


Fig. 5.32 Execution times for Intersection, Difference and Semi-join

For example, computing intersection of two relations with the cardinality of 1 k and a tuple size of 64 bytes takes 0.3 seconds. On twice as large relations it takes 0.6 seconds etc. The maximum possible performance increases linearly with the size (cost), of the Associative Array. This seems to be a very good result, but because we live in the world of sequential computers we must also look at the ESC.

Figure 5.33 shows ESC (in nanoseconds).

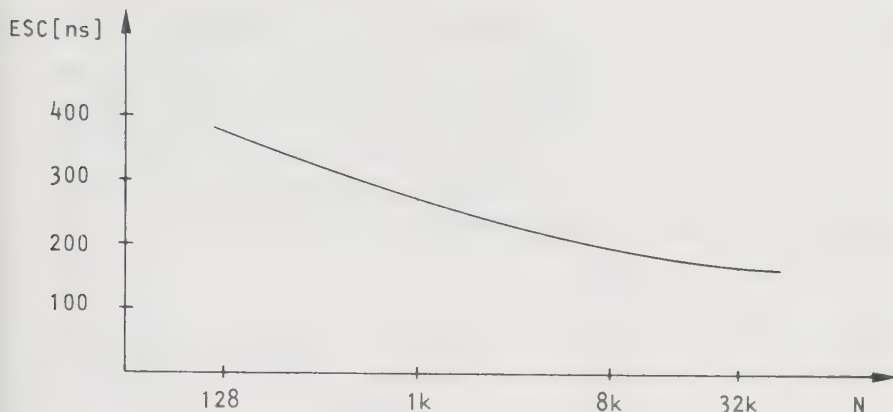


Fig. 5.33 ESC for Intersection, Difference, Semi-join

By examining Figure 5.33, two interesting observations can be made.

The first is that if a sequential computer is to outperform LUCAS (128 processors), it must make a comparison of two bytes in at most 375 nanoseconds. This is by itself not impossible for a minicomputer, but as we will discuss in Chapters 8 and 9, the overheads involved makes the total execution time much larger and the Associative Array would be to prefer.

The second observation is the following. If we increase the size of the Associative Array e. g. from 128 to 32k, it will be possible to process 256 times larger relations (in 256 times longer time). But, the same execution time on a sequential computer can be achieved by decreasing the time which is necessary for a byte comparison to only a half, which amounts to e.g. a doubling of the clock frequency.

PROJECTION

NR2 in the timing equation of the Projection can be expressed as $p \cdot N$, where p is the probability that a given tuple will belong to the result, and N is the cardinality of the operand relation. Figure 5.34 shows the execution time for $p=0.1$, $p=0.5$ and $p=1$ for the tuple size $b=64$. The cardinality of the operand relation is the same as the number of processors in the Associative Array.

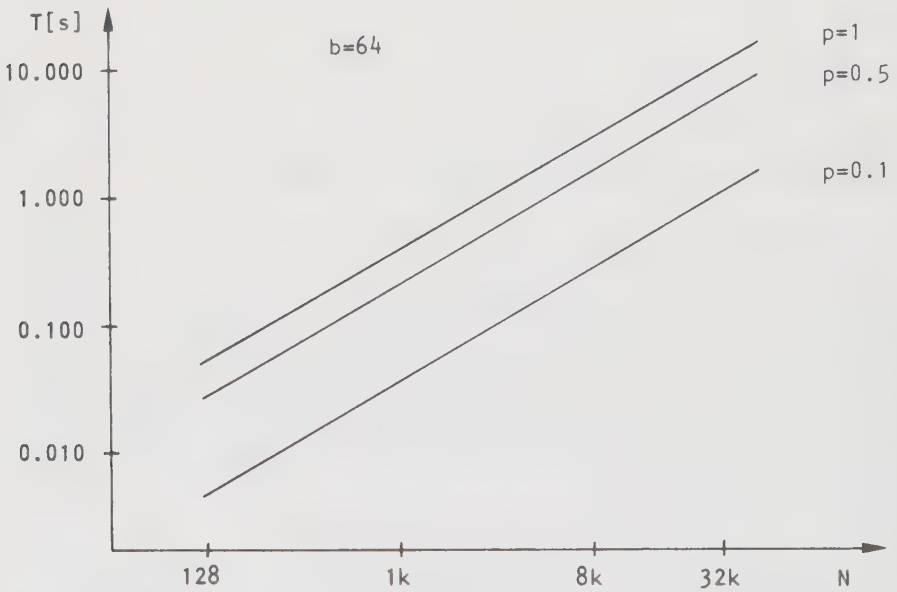


Fig. 5.34 Execution time of Projection

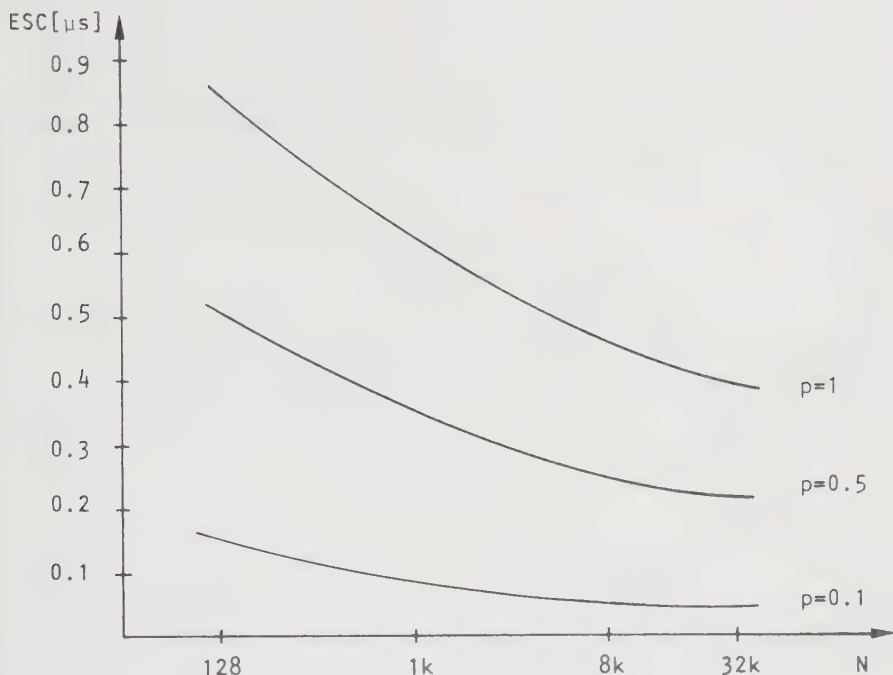


Fig. 5.35 ESC for Projection

Figure 5.35 indicates that for small p , an Associative Array perform much more cost effectively than sequential computers. For example, if a sequential computer is to perform the Projection in the situations where $p=0.1$ as fast as an Associative Array with 1k processors, it must be able to compare 2 bytes faster than 10^7 times/second.

We can see that for small p the Associative Array compared to a sequential computer is much better than when p is large. Similar result concerning the Join will be discussed in Chapter 8.

JOIN

We will discuss the Join operation in Chapter 8.

DIVISION

Figure 5.36 shows the execution time, and Figure 5.37 shows ESC of the Division operation for $b=16$ bytes and for $N2=0.5*N$ and $N2=0.1*N$

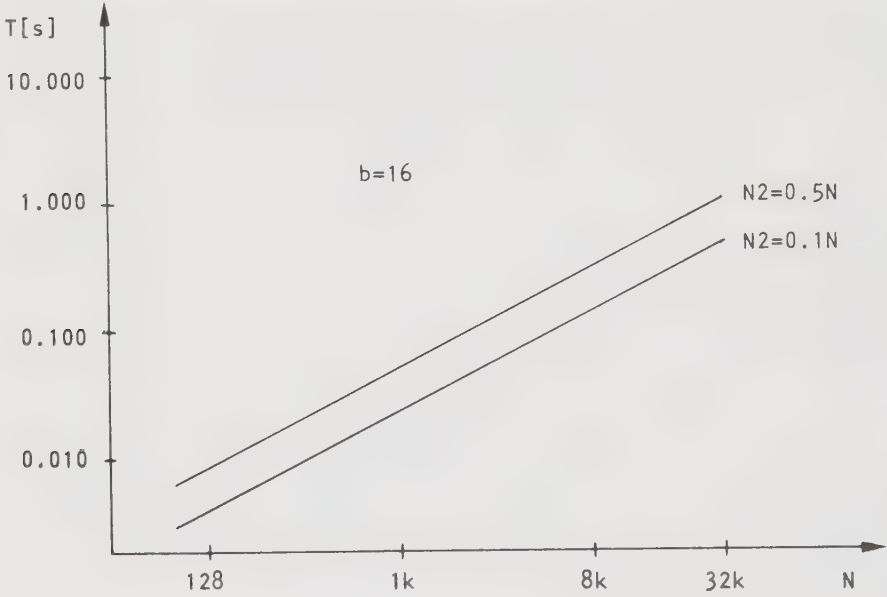


Fig. 5.36 Execution time of Division

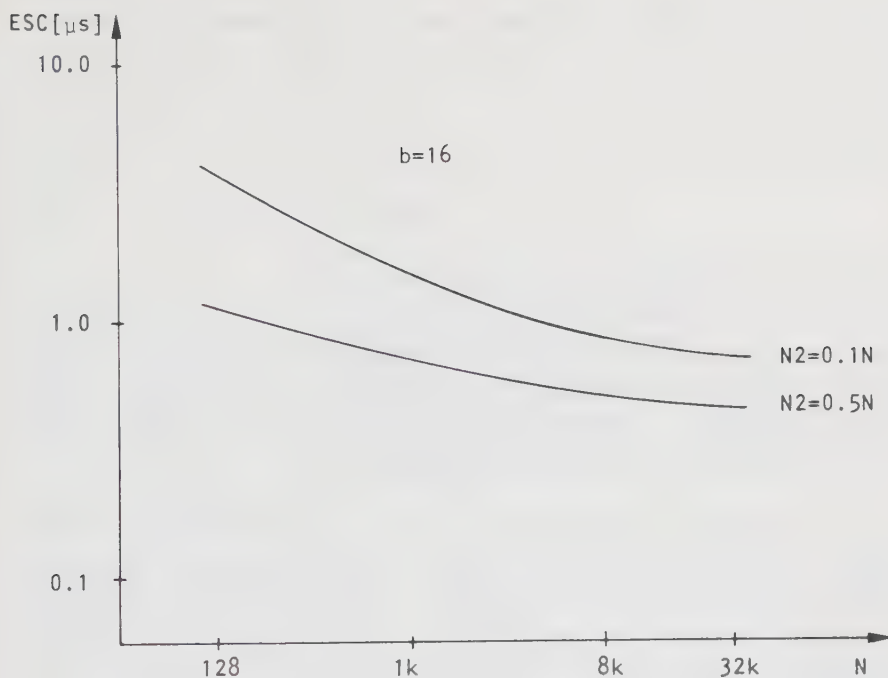


Fig. 5.37 ESC for Division

UNION

Both the timing equation and ESC are similar to those of the Projection.

PRODUCT

The advantage of using an Associative Array is that it can execute operations in parallel, e. g. by making comparisons one to many. The Product operation requires only a transfer of tuples and this transfer must be performed serially. In Table 5.2 we can see that the transfer of one byte calls for 22 clock cycles, thus with a 5 MHz clock frequency it makes approximately 4 microseconds. It is quite a slow rate when compared to execution on a sequential computer. But it does not mean that it is not a useful operation. In an Associative Array, the operands to be concatenated may be a result of some previous operation and they are identified associatively, by their Markbitslices. Also, the destination of the result of the Product operation, i.e. those memory words where the

result tuples will be assembled, can be determined associatively.

5.4.3 COMPARISON WITH ALTERNATIVE DESIGNS

There are only a few Associative Arrays of the same type as LUCAS on which timing data for relational operations are reported. We will compare LUCAS only with STARAN, RELACS, and with PAM of RDB. All these designs were presented in Chapter 4.

A search operation in STARAN [Berra79], which is equivalent to the Selection operation in LUCAS, takes $1 + 0.2 \cdot n$ microseconds where n is the number of bits in the argument. This is roughly equal to 1.6 microseconds/byte. The maximum size of the attribute is 256 bits. In LUCAS this operation takes 10 clock cycles, with a clock frequency of 5 MHz we obtain 2 microseconds/byte. STARAN is slightly faster. We can also compare the reading times. It takes 16 microseconds to read one 256-bit word from the array of STARAN, it is 0.5 microseconds/byte. In LUCAS, to copy one byte from a memory word into an I/O Register takes 8 clock cycles which is 1.6 microseconds. STARAN is 3 times faster. If a whole 128-bytes byteslice is to be read from LUCAS it takes $128 + 8$ clock cycles which is 0.2 microseconds/byte, and this make LUCAS twice as fast as STARAN.

This relative similarity of performance between STARAN and LUCAS with respect to I/O and search processing makes it possible to apply many conclusions about the usefulness of STARAN to LUCAS as well. For example, Berra and Oliver [Berra79] have convincingly demonstrated the great potential of STARAN in a database management environment.

In RELACS, which is a paper machine, an Associative Unit has an assumed size of between 1k and 100k words and a width of 1k bits. The search operation have an assumed speed half that of STARAN,

3.2 microseconds/byte, 1.6 times that of LUCAS. More complex operations, such as Join, are implemented in RELACS with the help of a Comparand Register Array, providing for parallel comparison many to many. In principle, this feature can be implemented [Digby73], but because of the complexity of connections we believe it will never be practically feasible.

Shaw's RDB is a paper machine. Its central part is a Primary Associative Memory, PAM. According to Shaw PAM could be realized with a large-scale distributed logic memory, or with a bit-serial or word serial design. It has a capacity of between 10k and 1M bytes (the capacity of LUCAS is $128 \times 512 = 64k$ bytes). The time for the Selection operation (called 'associative probe' by Shaw) is more than 0.8 microseconds/byte.

The capabilities of PAM are satisfied by LUCAS. For example a command of PAM such as

parallel set <flag> in all <tuple variable> of <relation> with <conditions>

corresponds to the Selection operation in LUCAS. A control structure

for each <tuple variable> with <conditions> set <flag> and do <statement>

of PAM can be implemented with the help of SELECT FIRST AND REMOVE in LUCAS.

This makes Shaws algorithms comparable to ours.

For example, in LUCAS, the number of searches needed for the Project operation is equal to the cardinality of the result relation. In PAM, the number of searches is twice the cardinality of the result relation. This is because one of the two searches in the main loop of Shaw's algorithm, is executed as a simple SELECT FIRST AND REMOVE operation on LUCAS. This makes our algorithm faster. Similar observations can be made when analyzing algorithms for the Intersection and Difference operations. While Shaw's algorithm for Join makes it necessary to perform search three times for each tuple

of one of the operand relations, our algorithm makes it necessary to do search only twice for each unique value of a joining attribute of one of the relations! The number of searches is thus considerably small on LUCAS than on PAM.

The last paragraph is shows the power of bit-serial processing. The possibility to select and process information about subsets is a very advantageous feature. It makes it simple to do logical operations on the results of searches according to different search criteria. In the second phase of the Division operation this technique was used iteratively.

Chapter 6.

INTERNAL QUERY EVALUATION

In Chapter 5, we have described algorithms implementing algebraic operations on relations loaded into the Associative Array of LUCAS. In the present chapter we will go a step further and demonstrate how these algorithms can be used for evaluating whole complex queries entirely within the Associative Array.

The evaluation method is based on decomposition of a query into a sequence of algebraic operations which are serially executed on relations in the Associative Array.

The method will be presented by eleven commented examples of evaluation of queries to a simple database.

As a general environment for query evaluation we will assume a system configuration shown in Figure 6.1. This configuration forms a typical database computer which consists of: a Supporting Processor, a Disk memory, a Console and an Associative Array with its Control Unit. We assume that the cardinality of relations necessary for answering a query is such that they fit inside the Associative Array.

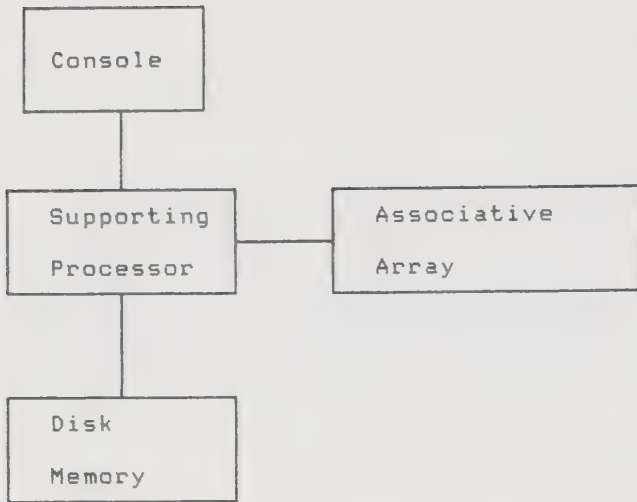


Fig. 6.1

The database computer operates in the following way. When a user types a query to the database in some query language on the Console the query is parsed and translated by the Supporting Processor into a sequence of algebraic operations. The Supporting Processor determines which relations are needed for answering the query and if they are not present in the Associative Array, it allocates a free area in the Associative Array and loads these relations from the disk memory. The Supporting Processor maintains two directories, a disk directory with information about relations on the disk and an array directory with information about relations currently in the Associative Array. The array directory provides the Supporting Processor with information about addresses, types and sizes of attributes, addresses of Workfields etc. The execution of the query is governed by a sequence of instructions issued by the Supporting Processor. Before each instruction, some necessary parameters must be sent to the Control Unit. The result of an instruction is always a new relation created in the Associative Array. The final relation in this sequence is the result of the query. This relation can subsequently be saved on the disk and added to the database or

submitted to some application program (computing e.g. averages) or it can simply be displayed on the Console.

The database computer must perform many other functions in addition to those described above, but we will concentrate our interest only on the use of the Associative Array in answering queries.

6.1 A TYPICAL DATABASE AND A SAMPLE OF QUERIES

To demonstrate our method we need a relational database which is simple enough to be easily understood but which has enough content to give the material for constructing and evaluating interesting queries. There is no better choice than what is probably the best-known relational database in the world, the Suppliers-Parts-Projects (S-P-P) database. This database consists of four relations and is described in An Introduction to Database Systems by C. J. Date [Date81, pp 114] where it is used in numerous exercises.

The S-P-P database, used by a hypothetical multinational corporation producing computer hardware, contains the information concerning a number of projects in which the corporation is involved. All the information available to users is represented in four relations S, P, J and SPJ.

Relation S, one instance of which is shown in Figure 6.2, contains the information about suppliers of different parts to current projects. It has four attributes: S# the unique supplier number, SNAME the suppliers' names, STATUS with the integers giving status of each supplier, and CITY with names of cities. The intended interpretation of the attribute CITY is the location of the suppliers. The key of this relation is the attribute S#.

S			
S#	SNAME	STAT	CITY
S1	SMITH	20	LONDON
S2	JONES	10	PARIS
S3	BLAKE	30	PARIS
S4	CLARK	20	LONDON
S5	ADAMS	30	ATHENS

Fig. 6.2 Relation S

Relation P, shown in Figure 6.3, contains the information about parts supplied to different projects. It has five attributes: P# the unique part number, PNAME the names of parts, COLOR the colours of parts, WEIGHT the integers giving the weights of parts, CITY names of cities. The intended interpretation of the attribute CITY is the location where the parts are stored. The key of this relation is the attribute P#.

P				
P#	PNAME	COLOR	WEIGHT	CITY
P1	NUT	RED	12	LONDON
P2	BOLT	GREEN	17	PARIS
P3	SCREW	BLUE	17	ROME
P4	SCREW	RED	14	LONDON
P5	CAM	BLUE	12	PARIS
P6	COG	RED	19	LONDON

Fig. 6.3 Relation P

Relation J, shown in Figure 6.4, contains the information about current projects. It has three attributes: J# the unique project numbers of each project, JNAME the names (probably covert) of projects, and CITY the names of cities. The intended interpretation of the attribute CITY is the location of plants where the projects are developed. The key of this relation is the attribute J#.

J		
J#	JNAME	CITY
J1	SORTER	PARIS
J2	PUNCH	ROME
J3	READER	ATHENS
J4	CONSOLE	ATHENS
J5	COLLATOR	LONDON
J6	TERMINAL	OSLO
J7	TAPE	LONDON

Fig. 6.4 Relation J

Relation SPJ, shown in Figure 6.5 connects the information about the specified suppliers supplying the specified parts to the specified projects in the specified quantity. It has four attributes: S# the supplier numbers (same as in relation S), P# the part numbers (same as in relation P), J# the project numbers (same as in relation J), and QTY the integers standing for delivered quantity. The key of this relation is the combination of the S#, P# and J# attributes.

SPJ			
S#	P#	J#	QTY
S1	P1	J1	200
S1	P1	J4	700
S2	P3	J1	400
S2	P3	J2	200
S2	P3	J3	200
S2	P3	J4	500
S2	P3	J5	600
S2	P3	J6	400
S2	P3	J7	800
S2	P5	J2	100
S3	P3	J1	200
S3	P4	J2	500
S4	P6	J3	300
S4	P6	J7	300
S5	P2	J2	200
S5	P2	J4	100
S5	P5	J5	500
S5	P5	J7	100
S5	P6	J2	200
S5	P1	J4	1000
S5	P3	J4	1200
S5	P4	J4	800
S5	P5	J4	400
S5	P6	J4	500

Fig. 6.5 Relation SPJ

As an example of how to interpret a tuple of the SPJ relation we can look at the first tuple which is $\langle S1, P1, J1, 200 \rangle$. It says that the supplier S1 has delivered 200 units of part P1 to project J1.

The connection between different relations in the S-P-P database is mediated by the fact that the attributes S#, P# and J# in the relation SPJ have the same domains as the key attributes in the relations S, P and J and also that the attributes called CITY in the S, P and J relation have values drawn from the same domain. The fact that the attributes have the same name in different relations is just incidental.

Queries to the S-P-P database, which we will evaluate in the next section, have been borrowed from Chapter 7 in Date's book. The queries are stated in English and as an exercise, the student of the book is asked to express them in different query languages. Date also gives solutions to the exercises. The queries are expressed in SQL, which is a data manipulation language of an experimental relational database management system, System R [Astrahan79], in Query By Example [Zloof75] with its special two dimensional syntax, in Relational Calculus and finally in Relational Algebra.

We select some representative queries from over 30 exercises. (The number in square brackets after each query refers to the number of the corresponding exercise in the book.)

- * 1) Get JNAME values for projects supplied by supplier S1. [7.6]
- * 2) Get S# values for suppliers who supply both projects J1 and J2. [7.8]
- * 3) Get S# values for suppliers who supply project J1 with a red part. [7.9]
- * 4) Get P# values for parts supplied to any project in London. [7.10]
- * 5) Get S# values for suppliers who supply a London or Paris project with a red part. [7.11]
- * 6) Get P# values for parts supplied to any project by a supplier in the same city. [7.12]
- * 7) Get J# values for projects not supplied with any red part by any London supplier. [7.15]
- * 8) Get S# values for suppliers supplying at least one part supplied by at least one supplier who supplies at least one red part. [7.16]

- * 9) Get S# values for suppliers who supply the same part to all projects. [7.21]
- * 10) Get J# values for projects which use only parts which are available from supplier S1. [7.25]
- * 11) Get J# values for projects supplied by supplier S1 with all parts that supplier S1 supplies. [7.26]

6.2 EVALUATION OF THE QUERIES IN THE ASSOCIATIVE ARRAY

A query is executed as a series of operations on relations in the Associative Array. Each operation creates a new, temporary, relation from one or two old relations. The final relation in this sequence is the result of the query. The execution is guided by the Supporting Processor issuing instructions and their parameters (e.g. addresses to attributes and their sizes) to the Control Unit of the Associative Array. In the case of the Selection operation, a value to be compared with the contents of some attribute of a relation in the Associative Array is loaded into the Comparand Register.

We demonstrate our method by way of examples, showing how a number of selected queries to the S-P-P database are executed. Each query will first be stated in English, then in the formalism of the Relational Algebra language of Date [Date81] and finally in the notation of the Algebraic Assignment language defining the sequence of operations needed for execution of the query. An accompanying figure will display all relevant original relations and all temporary relations created during the execution. The Markbitslices will be given to the right of the relations, whereas their real position in the Associative Array is determined by the Supporting Processor.

To understand how the information in the figures should be interpreted we can study Figure 6.6. It shows data of three relations J, T1 and T2 in the Associative Array together with their associated Markbitslices. Above the relations we indicate which attributes belong to which relations. Relations J and T1 have three attributes (J#, JNAME, CITY) each, and relation T2 has one attribute (JNAME). Relation J is one of the original relations of the S-P-J database, T1 is a derived relation obtained by the operation Selection on J where the value of the attribute CITY is LONDON. T2 is the result of the Projection of T1 on the attribute JNAME, it only consists of two tuples with the values TAPE and COLLATOR. The information about names of relations, addresses to attributes and Workfields is maintained by the Supporting Processor.

T2			
T1			
J			
J#	JNAME	CITY	TTJ
J1	SORTER	PARIS	1
J2	PUNCH	ROME	1
J3	READER	ATHENS	1
J4	CONSOLE	ATHENS	1
J5	COLLATOR	LONDON	111
J6	TERMINAL	OSLO	1
J7	TAPE	LONDON	111

Fig. 6.6 Relations J, T1, and T2

QUERY 1

Get JNAME values for projects supplied by supplier S1. [7.6]

There is no relation connecting the supplier numbers with the names of the projects which they supply. Hence, the information from two relations, SPJ connecting the supplier numbers with the project numbers and J connecting the project numbers with the project names, is needed. To answer the query, the numbers of the projects supplied by supplier S1 must be extracted from SPJ and used in J to look up the names of those projects. A way to solve this query is suggested by the following algebraic expression:

$(J \text{ JOIN } (SPJ \text{ WHERE } S\# = 'S1') [J\#]) [JNAME]$

First, tuples with S1 as the value of the attribute S# are selected from SPJ. Then the numbers of projects supplied by S1 are obtained by the Projection on the attribute J#; the project numbers are then combined with J by the Join and after the Projection on the attribute JNAME the result of the query, the names of the projects, is obtained.

In the Algebraic Assignment language notation, the execution of the query in the Associative Array consists of the following four steps:

- 1) $T1 := SPJ \text{ WHERE } S\# = 'S1'$
- 2) $T2 := T1 [J\#]$
- 3) $T3 := J \text{ SEMIJOIN } T2 \text{ ON } J\#$
- 4) $T4 := T3 [JNAME]$

Figure 6.7 illustrates the execution. In step 1, the value S1 is loaded into the Comparand Register above the attribute S# of SPJ and the Selection is performed, creating the relation T1. Relation T1 consists of tuples of SPJ pointed out by the Markbitslice of T1. In step 2, the Projection2 of T1 on the attribute J# is performed creating the relation T2. In step 3, relation T3 is created by the Semi-join between J and T2 on the attribute J# with a domain common to both relations. Data of relation T3 are physically a subset of J.

In step 4, the result of the query, the relation T4, is produced by the Projection2 of T3 on the attribute JNAME.

T2				T4		
T1				T3		
SPJ				J		
S#	P#	J#	QTY	J#	JNAME	CITY
S1	P1	J1	200	J1	SORTER	PARIS
S1	P1	J4	700	J2	PUNCH	ROME
S2	P3	J1	400	J3	READER	ATHENS
S2	P3	J2	200	J4	CONSOLE	ATHENS
S2	P3	J3	200	J5	COLLATOR	LONDON
S2	P3	J4	500	J6	TERMINAL	OSLO
S2	P3	J5	600	J7	TAPE	LONDON
S2	P3	J6	400			
S2	P3	J7	800			
S2	P5	J2	100			
S3	P3	J1	200			
S3	P4	J2	500			
S4	P6	J3	300			
S4	P6	J7	300			
S5	P2	J2	200			
S5	P2	J4	100			
S5	P5	J5	500			
S5	P5	J7	100			
S5	P6	J2	200			
S5	P1	J4	1000			
S5	P3	J4	1200			
S5	P4	J4	800			
S5	P5	J4	400			
S5	P6	J4	500			

Fig. 6.7 Execution of Query 1

The result of Query 1 is the relation T4 with one attribute, JNAME, consisting of two tuples, <SORTER> and <CONSOLE>.

The execution of Query 1 called for creating four temporary relations from two original relations. But the only new space in the Associative Array used during the execution was the space used by the four Workfields of temporary relations. Four algebraic operations, one Selection, two Projection2 and one Semi-join were executed.

When executing a query in the Associative Array, the Join should be avoided because its execution is time demanding and the result of the Join may sometimes occupy a space in the Associative Array much larger than the argument relations. In this query, and as a matter of fact in most queries (as we will see later), the Join used in the

Relational Algebra expression can be substituted by the Semi-join with the attractive consequence that the result of the Semi-join is physically a subset of one of the argument relations and no new space in the Associative Array needs to be allocated for the result data.

QUERY 2

Get S# values for suppliers who supply both projects J1 and J2.
[7.8]

All the information needed to answer this query can be found in one relation, SPJ, which connects, among other things, supplier numbers with project numbers. The algebraic expression solving the query is straightforward:

```
(SPJ WHERE J#='J1')[S#]
INTERSECT
(SPJ WHERE J#='J2')[S#]
```

In the Algebraic Assignment language notation, the execution of the query consists of the following five steps:

- 1) T1:= SPJ WHERE J#='J1'
- 2) T2:= SPJ WHERE J#='J2'
- 3) T3:= T1[S#]
- 4) T4:= T2[S#]
- 5) T5:= T3 INTERSECTBY T4

Figure 6.8 illustrates the execution. In step 1, the value J1 is loaded into the Comparand Register above the attribute S# in SPJ and the Selection is performed creating the relation T1. In a similar way, in step 2, relation T2 including the tuples with the information about the project J2 is created. In steps 3 and 4, the Projection2 of T3 and T4 on the attribute S# is performed. In the final step 5

the information from relations T3 and T4, the numbers of the suppliers supplying project J1 and J2 respectively, is compared by the Intersection and the result of the query, relation T5, is produced.

T5																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																								</
----	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	----

Fig. 6.8 Execution of Query 2

Five temporary relations were created during the execution of this query. After the two Selection and the two Projection2 operations all relevant data from the SPJ relation were reduced to two relations T3 and T4. The data of T3 and T4 are physically a part of the original relation SPJ, they even belong to the same attribute of SPJ, but still T3 and T4 are two different relations. T3 and T4 are compared in step 5 by the Intersection.

QUERY 3

Get S# values for suppliers who supply project J1 with a red part.
[7.9]

The information needed to answer this query can be found in relation P with the parts and their colours and in relation SPJ, with the part numbers, the supplier numbers and the project numbers. The algebraic expression solving the query is the following:

((SPJ WHERE J#='J1') JOIN (P WHERE COLOR='RED'))[S#]

In the Algebraic Assignment language notation, the execution of the query consists of the following five steps.

I)

II)

1) T1:= P WHERE COLOR='RED'

2) T2:= T1[P#] ;key

3) T3:= SPJ WHERE J#='J1'

T3:= SPJ SEMIJOIN T2 ON P#

4) T4:= T3 SEMIJOIN T2 ON P#

T4:= T3 WHERE J#='J1'

5) T5:= T4[S#]

Figure 6.9 illustrates the execution. In step 1, the value RED is loaded into the Comparand Register above the attribute COLOR in P and the Selection is performed creating the relation T1. In step 2, the relation T2 is created as a result of the Projection1 of T1 on P#. Because P# is a key attribute in T1 there is no need to eliminate redundant tuples. The only action performed in the Associative Array is to copy the Markbitslice of T1 into the Markbitslice of T2. There are two alternative ways to continue this computation. According to the first alternative (I), in step 3, the value J1 is loaded into the Comparand Register above J# in SPJ and the Selection is performed creating the relation T3. In step 4 the relation T4 is created by the Semi-join between T3 and T2 on the attribute P# common to both relations. According to the second alternative (II), in step 3, the relation T3 is created by the Semi-join between SPJ and T2 on the attribute P# with a domain common to both relations. In step 4, the value J1 is loaded into the

Figure 6.9 we can see that the temporary relation T3, created in the computation according to the second alternative (II) is much larger than T3 in the first alternative (I). The total time for the execution of the query is the same in both cases because the execution time for the Selection in step 3 in (I) respective 4 in (II) is independent of the number of tuples in the argument relation and the Semi-join on an attribute in step 4 in (I) and 3 in (II) is dependent only on the number of tuples in T2. Also, the space in the Associative Array occupied by data is the same for both alternatives. The algebraic expression of the query suggests making both SPJ and P smaller before doing the Join, as was done in the first alternative, and this would be the obvious solution in the case of a sequential execution on a von Neumann computer but in the Associative Array it has no advantage.

QUERY 4

Get P# values for parts supplied to any project in London. [7.10]

The information needed to answer this query can be found in relation J, with the project numbers and the names of cities where the projects are located, and in relation SPJ, with the project numbers and the part numbers. The algebraic expression solving this query is the following:

$(SPJ \text{ JOIN } (J \text{ WHERE CITY}='LONDON'))[P\#]$

In the Algebraic Assignment language notation, the execution consists of the following four steps:

- 1) T1:= J WHERE CITY='LONDON'
- 2) T2:= T1[J#] ;key
- 3) T3:= SPJ SEMIJOIN T2 ON J#
- 4) T4:= T3[P#]

Figure 6.10 illustrates the execution. In step 1, the value LONDON is loaded into the Comparand Register above the attribute JNAME of J and the Selection is performed giving T1. Step 2 is the Projection1 of T1 on the key attribute J# giving T2. In step 3, the relation T3 is created by the Semi-join between SPJ and T2 on the attribute J# with its domain common to both relations and the result of the query, the relation T4, is obtained in step 4 from T3 by the Projection2 on the attribute P#.

The first projection in step 2 is the fast Projection1 because the projected attribute is the key attribute in T1.

T4							
T3							
SPJ							
S#	P#	J#	QTY				
S1	P1	J1	200				
S1	P1	J4	700				
S2	P3	J1	400				
S2	P3	J2	200				
S2	P3	J3	200				
S2	P3	J4	500				
S2	P3	J5	600				
S2	P3	J6	400				
S2	P3	J7	800				
S2	P5	J2	100				
S3	P3	J1	200				
S3	P4	J2	500				
S4	P6	J3	300				
S4	P6	J7	300				
S5	P2	J2	200				
S5	P2	J4	100				
S5	P5	J5	500				
S5	P5	J7	100				
S5	P6	J2	200				
S5	P1	J4	1000				
S5	P3	J4	1200				
S5	P4	J4	800				
S5	P5	J4	400				
S5	P6	J4	500				

T2					
T1					
J					
J#	JNAME	CITY			
J1	SORTER	PARIS			
J2	PUNCH	ROME			
J3	READER	ATHENS			
J4	CONSOLE	ATHENS			
J5	COLLATOR	LONDON			
J6	TERMINAL	OSLO			
J7	TAPE	LONDON			

Fig. 6.10 Execution of Query 4.

QUERY 5

Get S# values for suppliers who supply a London or Paris project with a red part. [7.11]

The information needed to answer this query can be found in three relations. In P the information about the part numbers and their colour is stored, J contains the project numbers and the names of the cities where the projects are located. SPJ finally, connects the project numbers, the part numbers and the supplier numbers. The algebraic expression solving the query is the following:

```
((J WHERE CITY='LONDON' OR CITY='PARIS')[J#]
  JOIN SPJ
    JOIN (P WHERE COLOR='RED')[P#])[S#]
```

In the Algebraic Assignment language notation, the execution consists of the following nine steps:

- 1) T1:= P WHERE COLOR='RED'
- 2) T2:= T1[P#] ;key
- 3) T3:= J WHERE CITY='PARIS'
- 4) T4:= J WHERE CITY='LONDON'
- 5) T5:= T3 UNION T4
- 6) T6:= T5[J#] ;key
- 7) T7:= SPJ SEMIJOIN T2 ON P#
- 8) T8:= T7 SEMIJOIN T6 ON J#
- 9) T9:= T8[S#]

Figure 6.11 illustrates the execution. In step 1, the value RED is loaded into the Comparand Register above the attribute COLOR of the relation P and the Selection is performed, giving T1. Step 2 gives T2 after the Projection1 of T1 on the key attribute P#. Steps 3, 4, 5 and 6 produce the relation T6 consisting of the numbers of the projects in Paris or London. Step 5 is the Union of the two relations T3 and T4. Because both T3 and T4 were created from the same original relation, no data has to be moved. T5 is created by logical OR between the Markbitslices of T3 and T4. Step 7 is the

Semi-join between SPJ and T2 on the attribute P# with a domain common to both relations producing T7. T7 consists of tuples from SPJ with the numbers of the red parts. In step 8, the Semi-join is performed again between the result of the previous Semi-join, T7, and T6, producing the relation T8. T8 includes the tuples from SPJ with the project numbers of the London or Paris projects supplied with red parts. Finally, in step 9, the Projection2 on the attribute S# yields the result of the query, the relation T9.

In the execution of this query, data extracted from two relations, P and J, are used in two consecutive Semi-joins on data from the third relation SPJ, reducing successively the set of tuples with relevant information.

Steps 3, 4 and 5 represent the Selection with the complex criterion PARIS OR LONDON and replace the expression

T:= J WHERE CITY='PARIS' OR CITY='LONDON'

This is an illustration of how complex operations can be avoided by using a sequence of simpler operations.

T2				
T1				
P				
P#	PNAME	COLOR	WEIGHT	CITY
P1	NUT	RED	12	LONDON
P2	BOLT	GREEN	17	PARIS
P3	SCREW	BLUE	17	ROME
P4	SCREW	RED	14	LONDON
P5	CAM	BLUE	12	PARIS
P6	COG	RED	19	LONDON

TTP
21111
1
1
111
1
111

T9			
T8			
T7			
SPJ			
S#	P#	J#	QTY
S1	P1	J1	200
S1	P1	J4	700
S2	P3	J1	400
S2	P3	J2	200
S2	P3	J3	200
S2	P3	J4	500
S2	P3	J5	600
S2	P3	J6	400
S2	P3	J7	800
S2	P5	J2	100
S3	P5	J1	200
S3	P4	J2	500
S4	P6	J3	300
S4	P6	J7	300
S4	P2	J2	200
S4	P2	J4	100
S4	P5	J5	500
S4	P5	J7	100
S4	P6	J2	200
S4	P1	J4	1000
S4	P3	J4	1200
S4	P4	J4	800
S4	P5	J4	400
S4	P6	J4	500

S
TTTT
987J1111
11
1
1
1
1
1
1
1
1
1
1
11
11
1111
1
1
1
1
1
11
11

T6		
T5		
T4		
T3		
J		
J#	JNAME	CITY
J1	SORTER	PARIS
J2	PUNCH	ROME
J3	READER	ATHENS
J4	CONSOLE	ATHENS
J5	COLLATOR	LONDON
J6	TERMINAL	OSLO
J7	TAPE	LONDON

TTTTJ
654311 11
1
1
1
111 1
1
111 1

Fig. 6.11 Execution of Query 5

QUERY 6

Get P# values for parts supplied to any project by a supplier in the same city. [7.12]

The information needed to answer this query can be found in three relations. In S the information about the supplier numbers and the names of the cities where suppliers reside is available, in J, the project numbers and the names of the cities where the projects are located and in SPJ, the project numbers, the part numbers and the supplier numbers. The algebraic expression solving the query is the following:

$(J[CJ\#,CITY] \text{ JOIN } SPJ \text{ JOIN } S[S\#,CITY])[P\#]$

In the Algebraic Assignment language notation, the execution consists of the following five steps:

- 1) $T1 := S[S\#,CITY]$;key
- 2) $T2 := J[CJ\#,CITY]$;key
- 3) $T3 := T1 \text{ JOIN}_2 T2 \text{ ON } CITY$
- 4) $T4 := SPJ \text{ SEMIJOIN } T3 \text{ ON } [S\#,J\#]$
- 5) $T5 := T4[P\#]$

Figure 6.12 illustrates the execution. In step 1, the Projection1 of the relation S on the two attributes S# and CITY gives T1. In step 2, T2 is created as the result of the Projection1 of J on the two attributes J# and CITY. Because there is no relation connecting the numbers of the suppliers and the projects in the same city, and this information is needed in deriving the answer of the query, such a relation must be created. In step 3, the Join2 on T1 and T2 on their attribute CITY creates this relation, called T3. (This operation was possible since elements of the attribute CITY in both relations are drawn from the same domain, the fact that both attributes have the same name, CITY, is just accidental and has no relevance.) The attribute CITY is not a part of T3 for the reason that it is not necessary in the further execution of the query. In step 4, the Semi-join of SPJ and T3 on the two attributes S# and

QUERY 7

Get J# values for projects not supplied with any red part by any London supplier. [7.15]

The information from all four relations is needed to answer this query. In S the information about the supplier numbers and the names of the cities is given, in P the part numbers and the part colour are located, J gives the project numbers of all current projects and finally, SPJ connects the part numbers, the project numbers and the supplier numbers. The algebraic expression solving this query is the following:

```
J[CJ#] MINUS
  (((S WHERE CITY='LONDON')[C#]
    JOIN SPJ
      JOIN (P WHERE COLOR='RED')[P#])[CJ#]
```

In the Algebraic Assignment language notation, the execution consists of the following nine steps:

- 1) T1:= S WHERE CITY='LONDON'
- 2) T2:= S[C#] ;key
- 3) T3:= P WHERE COLOR='RED'
- 4) T4:= T3[P#] ;key
- 5) T5:= SPJ SEMIJOIN T2 ON S#
- 6) T6:= T5 SEMIJOIN T4 ON P#
- 7) T7:= T6[CJ#]
- 8) T8:= J[CJ#] ;key
- 9) T9:= T8 DIFFERENCE T7

Figure 6.13 illustrates the execution. Steps 1 and 2 produce from S the relation T2 consisting of the supplier numbers of the suppliers in London. Steps 3 and 4 produce T4 consisting of the part numbers of the red parts. Steps 5 and 6 produce T6 with tuples from SPJ including the information about the London suppliers of red parts. The Projection2 of T6 on the attribute J# in step 7 gives T7 consisting of the project numbers of the projects supplied by the London suppliers with red parts. Step 8 gives T8 from J by the

Projection1 on the attribute J#. T8 contains the project numbers of all current projects. Finally, in step 9 the Difference between T8 and T7 creates the result relation T9.

The execution of this query is an example of the use of the Difference operation. The result of the query, the relation T9, is obtained by subtracting relation T7 which consists of the numbers of the projects supplied with a red part by a London supplier, from relation T8 which consists of the numbers of all current projects.

A question could arise whether there is really a need to use J to obtain the project numbers of all current projects in the relation T8, as seen in step 8, instead of doing the Projection2 of SPJ on J#. The answer is that in this particular example the result would be the same, but because the database is a dynamic entity, it could happen that there are projects in J about which there is no information in SPJ; possibly because no parts have been delivered to them yet.

T2					
T1					
S					
S#	SNAME	STAT	CITY		TTS 21
S1	SMITH	20	LONDON		111
S2	JONES	10	PARIS		1
S3	BLAKE	30	PARIS		1
S4	CLARK	20	LONDON		111
S5	ADAMS	30	ATHENS		1

T4						
T3						
P						
P#	PNAME	COLOR	WEIGHT	CITY	TTT	43
P1	NUT	RED	12	LONDON	111	
P2	BOLT	GREEN	17	PARIS	1	
P3	SCREW	BLUE	17	ROME	1	
P4	SCREW	RED	14	LONDON	111	
P5	CAM	BLUE	12	PARIS	1	
P6	COG	RED	19	LONDON	111	

T7			
T6			
T5			
SPJ			
S#	P#	J#	QTY
S1	P1	J1	200
S1	P1	J4	700
S2	P3	J1	400
S2	P3	J2	200
S2	P3	J3	200
S2	P3	J4	500
S2	P3	J5	600
S2	P3	J6	400
S2	P3	J7	800
S2	P5	J2	100
S3	P3	J1	200
S3	P4	J2	500
S3	P6	J3	300
S4	P6	J7	300
S5	P2	J2	200
S5	P2	J4	100
S5	P5	J5	500
S5	P5	J7	100
S5	P6	J2	200
S5	P1	J4	1000
S5	P3	J4	1200
S5	P4	J4	800
S5	P5	J4	400
S5	P6	J4	500

T9			
T8			
J			
J#	JNAME	CITY	TTJ 98
J1	SORTER	PARIS	11
J2	PUNCH	ROME	111
J3	READER	ATHENS	11
J4	CONSOLE	ATHENS	11
J5	COLLATOR	LONDON	111
J6	TERMINAL	OSLO	111
J7	TAPE	LONDON	11

Fig. 6.13 Execution of Query 7

QUERY 8

Get S# values for suppliers supplying at least one part supplied by at least one supplier who supplies at least one red part. [7.16]

The information needed to answer this query can be found in two relations. In P the information about the part numbers and the colours of the parts is stored, and in SPJ is the information connecting the supplier numbers with the part numbers. The algebraic expression solving the query is the following:

```
((((SPJ JOIN (P WHERE COLOR='RED')[P#])[S#])
      JOIN SPJ)[P#] JOIN SPJ)[S#]
```

In the Algebraic Assignment language notation, the execution consists of the following eight steps:

- 1) T1:= P WHERE COLOR='RED'
- 2) T2:= T1[P#] ;key
- 3) T3:= SPJ SEMIJOIN T2 ON P#
- 4) T4:= T3[S#]
- 5) T5:= SPJ SEMIJOIN T4 ON S#
- 6) T6:= T5[P#]
- 7) T7:= SPJ SEMIJOIN T6 ON P#
- 8) T8:= T7[S#]

Figure 6.14 illustrates the execution. Steps 1 and 2 produce from the relation P the relation T2 consisting of the part numbers of the red parts. Steps 3 and 4 produce from SPJ and T2 the relation T4 consisting of the supplier numbers of the suppliers supplying the red parts. Steps 5 and 6 produce from SPJ and T4 the relation T6 consisting of the part numbers of all parts supplied by all suppliers who supply red parts. Finally, steps 7 and 8 give the result relation T8 by the Semi-join of SPJ and T6 on S# and the subsequent Projection2 on S#.

T2		T1			TTP 21
P					
P#	PNAME	COLOR	WEIGHT	CITY	
P1	NUT	RED	12	LONDON	111
P2	BOLT	GREEN	17	PARIS	1
P3	SCREW	BLUE	17	ROME	1
P4	SCREW	RED	14	LONDON	111
P5	CAM	BLUE	12	PARIS	1
P6	COG	RED	19	LONDON	111

T8				S TTTTTTP 876543J
T7				
		T6		
T5				
T4				
T3				
SPJ				
S#	P#	J#	QTY	
S1	P1	J1	200	11111111
S1	P1	J4	700	1 1 11
S2	P3	J1	400	11 1 1
S2	P3	J2	200	1 1 1
S2	P3	J3	200	1 1 1
S2	P3	J4	500	1 1 1
S2	P3	J5	600	1 1 1
S2	P3	J6	400	1 1 1
S2	P3	J7	800	1 1 1
S2	P5	J2	100	1 1 1
S3	P3	J1	200	1111 1
S3	P4	J2	500	1111111
S4	P6	J3	300	1111111
S4	P6	J7	300	1 1 11
S5	P2	J2	200	111 1
S5	P2	J4	100	11 1 1
S5	P5	J5	500	111 1
S5	P5	J7	100	1 1 1
S5	P6	J2	200	1 1111
S5	P1	J4	1000	1 1 11
S5	P3	J4	1200	1 1 1
S5	P4	J4	800	1 1 11
S5	P5	J4	400	1 1 1
S5	P6	J4	500	1 1 11

Fig. 6.14 Execution of Query 8

In the execution of this query we can see that the relation SPJ is used three times by the Semi-join operation. The algebraic expression suggested to use three times the Join operation which is much more complex.

QUERY 9

Get S# values for suppliers who supply the same part to all projects.
[7.21]

The information needed to answer this query can be found in two relations. In J is the information about the project numbers of current projects and in SPJ is the information connecting the project numbers, the supplier numbers and the part numbers. The algebraic expression solving the query is the following:

$$(SPJ[S\#,P\#,J\#] \text{ DIVIDEBY } J[J\#])[S\#]$$

In the Algebraic Assignment language notation, the execution consists of the following four steps:

- 1) T1:= SPJ[S#,P#,J#] ;key
- 2) T2:= J[J#] ;key
- 3) T3:= T1 DIVIDEBY T2 ON J#
- 4) T4:= T3[S#]

Figure 6.15 illustrates the execution. In step 1, the relation T1 with each tuple representing a delivery, is created by the Projection1 of SPJ on a key consisting of the attributes S#, P# and J#. Step 2 gives the relation T2 with the project numbers of all projects. Step 3 is the Division of T1 by T2 on the attribute J#. The result of this division is the relation T3 with a combination of supplier numbers and part numbers for each supplier delivering the same part to all projects. Finally, in step 4, the result relation T4 is obtained after the operation Projection2 of T3 on the attribute S#.

The execution of this query is an example of the use of the Division operation.

T4							
T3							
T1							
SPJ				T2			
S#	P#	J#	QTY	J			TJ
S1	P1	J1	200	J#	JNAME	CITY	2
S1	P1	J4	700	J1	SORTER	PARIS	11
S2	P3	J1	400	J2	PUNCH	ROME	11
S2	P3	J2	200	J3	READER	ATHENS	11
S2	P3	J3	200	J4	CONSOLE	ATHENS	11
S2	P3	J4	500	J5	COLLATOR	LONDON	11
S2	P3	J5	600	J6	TERMINAL	OSLO	11
S2	P3	J6	400	J7	TAPE	LONDON	11
S2	P3	J7	800				
S2	P5	J2	100				
S3	P3	J1	200				
S3	P4	J2	500				
S4	P6	J3	300				
S4	P6	J7	300				
S5	P2	J2	200				
S5	P2	J4	100				
S5	P5	J5	500				
S5	P5	J7	100				
S5	P6	J2	200				
S5	P1	J4	1000				
S5	P3	J4	1200				
S5	P4	J4	800				
S5	P5	J4	400				
S5	P6	J4	500				

Fig. 6.15 Execution of Query 9

QUERY 10

Get J# values for projects which use only parts which are available from supplier S1. [7.25]

The information needed to answer this query can be found in three relations. In J is the information about the project numbers of all current projects, in P is the information about the part numbers, and SPJ connects the supplier numbers, the part numbers and the project numbers. The algebraic expression solving the query is the following:

```
J[J#] MINUS
  ((SPJ JOIN (PCP#] MINUS
    (SPJ WHERE S#='S1')[P#]))[J#])
```

In the Algebraic Assignment language notation, the execution consists of the following 9 steps:

- 1) T1:= SPJ WHERE S#='S1'
- 2) T2:= T1[P#]
- 3) T3:= PCP#] ;key
- 4) T4:= T3 DIFFERENCE T2
- 5) T5:= SPJ[P#,J#]
- 6) T6:= T5 SEMIJOIN T4 ON P#
- 7) T7:= T6[J#]
- 8) T8:= J[J#] ;key
- 9) T9:= T8 DIFFERENCE T7

Figure 6.16 illustrates the execution. In steps 1 and 2, the relation T2 is created with the numbers of the parts supplied by S1. Steps 3 and 4 produce the numbers of the parts which are not supplied by S1. Steps 5, 6 and 7 give the relation T7 with the numbers of the projects using parts not available from S1. Finally, steps 8 and 9 produce the result relation T9.

T4																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																		</
----	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	----

T7			
T6			
T5			
T2			
T1			
SPJ			
S#	P#	J#	QTY
S1	P1	J1	200
S1	P1	J4	700
S2	P3	J1	400
S2	P3	J2	200
S2	P3	J3	200
S2	P3	J4	500
S2	P3	J5	600
S2	P3	J6	400
S2	P3	J7	800
S2	P5	J2	100
S3	P3	J1	200
S3	P4	J2	500
S4	P6	J3	300
S4	P6	J7	300
S5	P2	J2	200
S5	P2	J4	100
S5	P5	J5	500
S5	P5	J7	100
S5	P6	J2	200
S5	P1	J4	1000
S5	P3	J4	1200
S5	P4	J4	800
S5	P5	J4	400
S5	P6	J4	500

T9			
T8			
J			
J#	JNAME	CITY	
J1	SORTER	PARIS	
J2	PUNCH	ROME	
J3	READER	ATHENS	
J4	CONSOLE	ATHENS	
J5	COLLATOR	LONDON	
J6	TERMINAL	OSLO	
J7	TAPE	LONDON	

TTTTTSP			
7	6	5	21J
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	

Fig. 6.16 Execution of Query 10

An interesting feature of this execution is the way it proceeds. It starts in the relation SPJ. Then, data extracted from SPJ are used together with data from P to derive the relation T4, the data of which are physically shared with the relation P. Next, T4 is used against SPJ, which is thus accessed for a second time, by data it has helped to create, and finally the result of the query is defined

from the relation J.

QUERY 11

Get J# values for projects supplied by supplier S1 with all parts that supplier S1 supplies. [7.26]

All the information needed to answer this query is contained in relation SPJ. The algebraic expression solving the query is the following:

$$SPJ[CJ\#,S\#,P\#] \text{ DIVIDEBY } (SPJ \text{ WHERE } S\#='S1')[C\#,P\#]$$

In the Algebraic Assignment language notation, the execution consists of the following four steps:

- 1) $T1 := SPJ \text{ WHERE } S\#='S1'$
- 2) $T2 := T1[C\#]$
- 3) $T3 := SPJ[C\#,J\#]$
- 4) $T4 := T3 \text{ DIVIDEBY } T2 \text{ ON } P\#$

Figure 6.17 illustrates the execution. Steps 1 and 2 give the relation T2 with the numbers of all parts supplied by supplier S1. Step 3 prepares data for the Division performed in step 4.

				T4				
				T3				
				T2				
				T1				
SPJ								
S#	P#	J#	QTY	T	T	T	T	S
				4	3	2	1	J
S1	P1	J1	200	1	1	1	1	
S1	P1	J4	700	1	1			
S2	P3	J1	400	1		1		
S2	P3	J2	200	1		1		
S2	P3	J3	200	1		1		
S2	P3	J4	500	1		1		
S2	P3	J5	600	1		1		
S2	P3	J6	400	1		1		
S2	P3	J7	800	1		1		
S2	P5	J2	100	1		1		
S3	P3	J1	200				1	
S3	P4	J2	500	1		1		
S4	P6	J3	300	1		1		
S4	P6	J7	300	1		1		
S5	P2	J2	200	1		1		
S5	P2	J4	100	1		1		
S5	P5	J5	500	1		1		
S5	P5	J7	100	1		1		
S5	P6	J2	200	1		1		
S5	P1	J4	1000				1	
S5	P3	J4	1200				1	
S5	P4	J4	800	1		1		
S5	P5	J4	400	1		1		
S5	P6	J4	500	1		1		

Fig. 6.17 Execution of Query 11

The execution of this query is an example of how a relation can be divided by its own data.

6.3 DISCUSSION

We have demonstrated in eleven examples a method of using the Associative Array to answer a complete query to a database. The method which we have developed is very simple and straightforward. Relevant relations are loaded into the Associative Array once and the query is evaluated by a series of algebraic operations producing a sequence of new relations. The final relation in this sequence is the result of the query. We will now turn our attention to the relevance of the method.

Among many pitfalls which may face researchers in the field of computer architecture the most treacherous are:

- * To give an elegant solution to a problem which does not exist.
- * To give a solution to a real problem, when this solution is applicable only in a very restricted or uninteresting domain of the problem.
- * To give a solution to only one part of a real problem, when this solution is impossible to integrate with the solution of the rest of the problem.
- * To give 'yet another' solution, different from already known solutions but not giving any advantage.

(To be just, it must be said that even if a solution falls into one of the above categories this does not necessarily mean that the solution is totally without interest.)

We will now discuss our method in the light of the four pitfalls listed above. Hopefully we can show that we have avoided them.

The first pitfall is the irrelevance of the solution. Have we addressed a problem which needs a solution or have we just developed

a great method for dehydrating elephants?

The answer is simple. The fast and efficient evaluation of a query is one of the most important tasks performed by a DBMS. Using a conventional von Neumann computer with sequentially executed software is not always feasible. There is certainly a need for new solutions.

The second pitfall is the inapplicability of the solution. In which areas can our method be used? This question has two dimensions; the first one has to do with logic and the second with technology.

a) Which queries can be evaluated in the Associative Array? How representative were our examples? Can it be used for answering any query to a database or are there queries which can be formulated but can not be answered entirely within the Associative Array, and require that data be shuffled between the host, doing part of the processing, and the Associative Array, doing the rest? The answer is that any query can be evaluated in the Associative Array if by 'any query' we mean any query expressed in the language of relational algebra. The proof was given in Chapter 5 where we showed that the Associative Array is 'relationally complete' by demonstrating the implementation of all algebraic operations.

b) What is the size of relations for which our method can be used? Obviously, the method can be used only if the relations fit into the Associative Array. The size of the Associative Array of LUCAS, 128 processors each with 4096 bits of memory, is clearly not realistic. There are not very many relational databases with the cardinality of relations limited to 128 (most probably there are none). But due to the regularity and linearity of the structure of the Associative Array, in the near future it will be possible to build Arrays similar to LUCAS with thousands, maybe tens of thousands of processors, and then our method will be feasible.

The third pitfall is the lack of integrability. How well can our method be integrated into a database computer which must perform many other functions in addition to query evaluations?

Are there any functions in the database computer which will be impossible to implement if the database computer includes an Associative Array? We have found no indications of this, and we believe that incorporating an Associative Array into the database computer gives no technological or system organizational problems.

The fourth pitfall, finally, is poor performance. We must ask the question: What advantages and disadvantages has our method?

One advantage is the speed of evaluation of a query. The Associative Array can be seen as a high-level language architecture. A number of powerful set-oriented operations are implemented directly in the hardware and their execution is very fast because the algorithms take advantage of the parallelism in the Associative Array. Another factor speeding up the evaluation is the one-to-one correspondence between statements in the Algebraic Assignment language defining the query and the operation of the Associative Array. Because there is no need for layers of software making tests, translation, interpretation, iterations counts, etc, there is no software overhead slowing down the execution.

Another advantage is the opportunity for simple implementation of the user's views and access privileges. The only way to identify a tuple of a relation in the Associative Array is by using the Markbitslice. A number of users may use the same relations but, having been assigned different Markbitslices, they have access to only a subset of tuples.

A very advantageous feature of our method is that it saves space in the Associative Array. Except for the Join operation, operations executed during the evaluation of a query do not create new data. The result of an operation is just a new Markbitslice, and even the little space it occupies can be released after the Markbitslice is used by some subsequent operation and no longer is needed.

The main disadvantage of our method is the processing of Joins on relations. The execution of a Join operation is not only slow but it might be the case that its result is larger than both the source relations, with the unhappy consequence that the result cannot be stored in the Associative Array. Fortunately, there is a large class of queries which can be handled without requiring the Join operation, where the much faster Semi-join can be used instead. (Among the eleven examples of query evaluation the Join was only necessary in one.) A considerable amount of research work is done on the question of identifying this class [Bernstein 81A, Bernstein 81B].

Chapter 7

COMPARATIVE PERFORMANCE EVALUATION OF A DATABASE COMPUTER

In this chapter, we will evaluate the performance of a backend database computer which contains an Associative Array.

The assumed system, shown in Figure 7.1, consists of a host computer, a disk memory and an Associative Array. In the following, we will refer to the combination of the Associative Array and the disk as LUCAS.

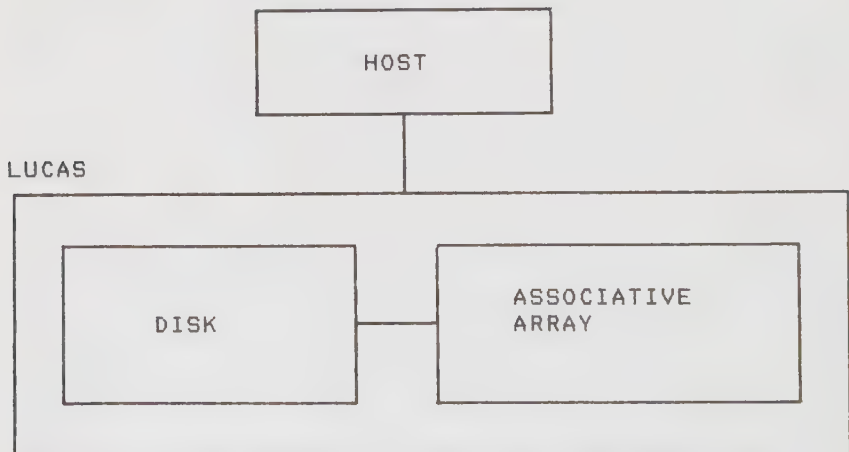


Fig. 7.1 Database computer

We will determine a response time of LUCAS to three benchmark retrieval queries. The times will be compared with response times of a number for different well-known database computer designs. We will use the results reported by Hawthorn and DeWitt [Hawthorn82], from

a performance analysis study of alternative database computers.

Hawthorn and DeWitt analyzed the performance of several database computers with respect to typical queries to a real database. Three classes of relational queries were identified: overhead-intensive, data-intensive, and multirelational queries. From each category one 'average query was selected and it was submitted to the following database computers: Associative Disks [Slotnik70], [Langdon78], RAP, CASSM, DBC, CAFS, and DIRECT [DeWitt79], and also to a conventional computer system with the INGRES relational database management system [Held75, Stonebraker76].

RAP, CASSM, DBC, and CAFS were described in Chapter 4.; DIRECT is a multiprocessor system consisting of simple query processors with the capacity of a microprocessor, and CCD memory cells interconnected by a crosspoint switch; Associative Disks represent in this study the earliest designs for enhancement of data management systems.

All database computers were assumed to function as backends to a host, a PDP 11/70. Each backend is a cellular system: data are stored in cells, with one processor per cell. Operations on the cells take place in parallel. The backends rely on the host to format the results for printing and to move the results to the user's terminal. Those backends which are not able to carry out arithmetic operations (included LUCAS) rely on the host to perform the arithmetic functions as well.

7.1 SPECIFICATION OF CHARACTERISTICS OF DATABASE MACHINES

Since all of the designs, except CAFS, are paper machines or rudimentary prototypes, Hawthorn and DeWitt made certain assumptions about their characteristics in order to make the performance comparisons fair and meaningful. We will make corresponding assumptions about LUCAS characteristics.

The data storage medium of all designs is assumed to be moving-head Ampex 9200 disk drives. Table 7.1 summarizes its parameters.

PARAMETER	MEANING	VALUE
BSIZE	block size	512 bytes
BTRACK	block/track	22 blocks
DROT	disk rotation time	16.7 ms
DAVAC	average access time	30 ms
DREAD	read time	0.8 ms/block
DCYL	blocks/cylinder	418 blocks
DTRACK	data tracks/cylinder	19

Tab 7.1 Disk parameters

Associative Disks, CASSM, DBC, and CAFS are assumed to have cell processors associated with read/write heads of the disks; RAP, DIRECT, and LUCAS are caching systems to which data must be loaded from the disks.

The size of the database computers were assumed to be the following: Associative Disks, CASSM, CAFS, and DBC contain 19 cell processors (one processor/track), RAP contains 16 cells with a capacity of 16k bytes each, and DIRECT contains 8 processors and 16 data cells with a capacity of 16k bytes each. The sizes indicate that we are discussing rather large systems.

Our assumption about LUCAS is that the size of the Associative Array is SIZE=2k processors and the clock frequency is 6 MHz. All other properties are the same as those of the real LUCAS. The width of a memory word of a processor in LUCAS is WIDTH=512 bytes. We will see that it is more than enough for storing the data necessary for answering our particular queries. Since data are read from the disk in 1.5 microseconds/byte, we assume that the time for loading data into the Associative Array is the same as the time for reading them from the disk. When tuples are serially read out

from the Associative Array, we assume that it takes $DO=2$ microseconds/byte (10 clock cycles = 8 for shifting byte into the I/O register + 2 for copying it into the I/O buffer register). A byte in the Comparand Register can be compared with a byteslice in all words in the Associative Array in $CPBC=2$ microseconds (10 clock cycles), a byte in a selected memory word can be compared with a byteslice in $CPB=6$ microseconds, and a byte in a selected word can be transferred into all other words in $TB=4$ microseconds.

The LUCAS parameters are summarized in Table 7.2.

PARAMETER	MEANING	VALUE
f	clock frequency	5 MHz
WIDTH	bytes in memory word	512 bytes
SIZE	processors	2048 processors
DI	data rate in	0.0015 ms/byte
DO	data rate out	0.002 ms/byte
CPBC	compare byte to CR	0.002 ms
CPB	compare byte in word	0.006 ms
TB	transfer byte	0.004 ms

Tab 7.2 LUCAS parameters

There is one further parameter to take into account in the timing equations for the total response time to a query. This is the host overhead time, HOV, which is due to query compilation and communication with the backend. HOV was carefully analyzed by Hawthorn and DeWitt who estimate that it is 0.042 s in the best case and 0.22 s in the worst case.

7.2 DATABASE AND QUERIES

The database for the three queries is the University of California at Berkeley Department of Electrical Engineering and Computer Science's course and room scheduling database. This database contains 24704 pages of data (12.6 Mbytes) in 102 relations. The data are information about courses taught: instructor's name, course name, room number, type of course, etc.

The queries are actual queries.

Query Q1 is representative for a class of short overhead-intensive queries.

```
Q1: retrieve(QTRCOURSE.day, QTRCOURSE.hour)
      where QTRCOURSE.instructor="despain,a.m."
```

The relation QTRCOURSE contains 1110 tuples. Each tuple has 24 attributes and is 127 bytes long. The relation is stored as a heap in 274 pages (blocks on disks). The attribute "day" is a character field, 7 bytes long; "hour" is 14 bytes long; the size of "instructor" is not specified in the paper, we assume that it is 30 bytes long.

In the test run at Berkeley, three tuples satisfied this query.

In the Algebraic Assignment language notation the query can be stated as:

```
T1:= QTRCOURSE WHERE INSTRUCTOR='despain,a.m.'
T2:= T1[DAY,HOUR]
```

It shall be assumed that the following algorithm is used in processing query Q1 in LUCAS:

- 1) The relation QTRCOURSE is loaded into the Associative Array. The size of the relation is such that the whole relation can be

loaded into the Array.

2) The values of the attribute "instructor" are compared in parallel with 'despain,a.m.' in a Comparand Register. As a result, three tuples are selected.

3) The values of "day" and "hour" of the selected tuples are output to the host. Since obviously each combination (day,hour,instructor) is unique, it is not necessary to check the result for redundancy.

Query Q2 is representative for a class of data-intensive multirelational queries.

```
Q2: retrieve(ROOMS.building, ROOMS.roomnum,
            ROOMS.capacity, COURSE.day, COURSE.hour)
      where ROOMS.roomnum=COURSE.roomnum and
            ROOMS.building=COURSE.building and
            ROOMS.type="lab"
```

The relation COURSE contains 11436 tuples in 2858 pages (1.4 Mbytes) with information about all the courses taught in the last four years. It requires 130 tracks (7 cylinders) of disk space. The relation ROOMS contains 282 tuples in 29 pages with information about every room that EECS Department can use for teaching courses. The (roomnum,building) attribute pair is 20 bytes long, the sizes of the attributes "capacity" and "type" is not specified, we assume they are 5 and 3 bytes long respectively.

There are 22 labs, and they were used 422 times in total. The result of this query is a list which contains the building, room number, capacity, day, and hour of use of any lab for the last four years.

In the Algebraic Assignment language notation the query can be stated as

```
T1:= ROOMS WHERE TYPE='lab'
```

```

T2:= T1[BUILDING,ROOMNUM,CAPACITY]
T3:= COURSE JOIN1 T2 OVER [BUILDING,ROOMNUM]
T4:= T3[BUILDING,ROOMNUM,CAPACITY,DAY,HOUR]

```

The algorithm used by LUCAS is the following:

- 1) The relation ROOMS is loaded into the Associative Array.
- 2) The 22 tuples with the information about labs are selected.
- 3) Cylinders of pages of the COURSE relation (1634 tuples each) are successively loaded from disks into the Associative Array, joined with the 22 tuples of the ROOMS relation and the result is output to the host. (This type of external Join evaluation will be discussed in Chapter 8.)

Query Q3 is representative for a class of data-intensive queries on a single relation. It includes an aggregate function.

```

Q3: retrieve(GMASTER.acct, GMASTER.fund,
            encumb=sum(GMASTER.encumb by
                       GMASTER.acct, GMASTER.fund))

```

The relation GMASTER contains 194 tuples in 97 pages. The sizes of the attributes are not specified, we assume that they are each 10 bytes long.

There are 17 unique values for the (acct,fund) pair. The query returns to the user the 17 unique (acct,fund) pairs along with their associated sums of values of the attribute "encumb".

The Algebraic Assignment language has no primitives for expressing aggregate functions.

This query can be executed in LUCAS in the following steps:

- 1) The relation GMASTER is loaded into the Associative Array.
- 2) By an operation similar to Projection, all tuples with identical (acct,fund) pairs are selected. Then, for each such partition, one (acct,fund) pair is output to the host together with "encumb" values of all tuples.
- 3) The host accumulates the sum of "encumb" values.

7.3 LUCAS RESPONSE TIMES

The response time to a query is the sum of the time spent in all components of the machine that cannot be overlapped.

Q1-Short query

The response time of LUCAS to query Q1, AAWORK, is given by

$$AAWORK = HOV + DAVAC + 274 * DREAD + AAPROC + AAOUT$$

where HOV is the host overhead time, DAVAC is the access time to the first track on disks with data of QTRCOURSE, $274 * DREAD$ is the time for reading 274 pages of the QTRCOURSE relation from the disk, AAPROC is the time spent on processing in the Associative Array and AAOUT is the time spent on returning the result to the host.

Since the size of the attribute "instructor" is 30 bytes, AAPROC is equal to

$$AAPROC = 30 * CPBC = 0.060 \text{ ms.}$$

Since the attribute "date" and "hour" are 21 bytes long together and since 3 tuples are read out,

$$AAOUT = 3 * 21 * DO = 0.126 \text{ ms.}$$

The worst case value of AAWORK is then

$$AAWORK = 0.22 + 0.03 + 274 * 0.0008 + 0.000060 + 0.000126 = 0.46 \text{ s}$$

Notice that the time spent on processing data in the Associative Array is negligible when compared to the overhead time or to the data transfer time.

The best case value of AAWORK is obtained if the Associative Array already holds the relation QTRCOURSE at the time when the query is issued and loading time is 0

$$AAWORK = 0.042 + 0.000060 + 0.000126 = 0.042 \text{ s.}$$

The situation that the relation is already in the Associative Array when a query is issued is in fact quite realistic. It is often the case that the same set of data is interrogated over and over again.

Q2-Multirelation query

The response time to query Q2 is given by:

$$\begin{aligned}
 AAWORK &= 2*HOV && \text{; host overhead, it is } 2*HOV \text{ because} \\
 & && \text{; two relations will be accessed} \\
 &+ DAVAC + 29*DREAD && \text{; time for loading ROOMS} \\
 &+ 7*(DAVAC + 412*DREAD) && \text{; time for loading 7 cylinders} \\
 & && \text{; of COURSE} \\
 &+ 3*CPBC && \text{; selection on ROOMS} \\
 &+ 7*(22*20*CPB) && \text{; 7 times join of 22 tuples of ROOM} \\
 & && \text{; with one cylinder load} \\
 &+ 7*(22*5*TB) && \text{; 7 times transfer of 22 values of} \\
 & && \text{; "capacity"} \\
 &+ 7*(60*50*D0) && \text{; 7 times output of result tuples} \\
 &= (2.71-3.07) \text{ s} && \text{; best case and worst case times,} \\
 & && \text{; depending on HOV}
 \end{aligned}$$

The last term gives the time spent on outputting the 422 tuples of the

result. We have assumed that they are uniformly distributed in the 7 COURSE loads, thus giving 60 tuples/load.

Q3-Aggregate Functions

The response time to query Q3 is given by:

AAWORK= HOV

+ DAVAC + 97*DREAD	;time for loading GMASTER
+ 17*20*CPB	;time to select 17 partitions
+ 17*20*D0	;time to output (acct,fund) pairs
+ 194*10*D0	;time to output encumb values
+ HDP	;time for computing 17 sums in the host

= (0.16-0.34) s + HDP

We estimate, rather conservatively that HDP, which is the time to compute 17 times the sum of 11 numbers is 0.01 s. Hence,

AAWORK= (0.16-0.34) s.

With the response times on LUCAS established, we can go on to performance comparisons.

7.4 PERFORMANCE COMPARISONS

Having established the response time of LUCAS to benchmark queries, we now proceed to compare the performance LUCAS with the performance of other systems.

* Its performance must be compared with the performance of a conventional database management system.

- * Its performance must be compared with other designs of database computers.

Response times to Q1, Q2, and Q3 for each system studied by Hawthorn and DeWitt and for LUCAS are plotted in Figures 7.2, 7.3, and 7.4.

The systems are ordered along the horizontal axis on the basis of "increasing complexity".

LUCAS has the largest number of processors - therefore it can be considered to be more complex than the other designs. The processors consist of RAM memory with simple bit-serial processing elements. They are much simpler than the cell processors in the other designs. The whole Associative Array may be implemented with a few chips (cf section 9.2) - therefore it can be argued that LUCAS is less complex than other designs. Thus, somewhat arbitrarily, we place LUCAS between CAFS and CASSM.



Fig. 7.2 Query Q1

Figure 7.2 shows that LUCAS exhibits the shortest best case time of all studied systems. It is also 3 times faster than INGRES. In the worst case the performance of LUCAS, though approximately the same as that of the other systems is actually worse than the performance of a conventional database management system. In processing query Q1, INGRES uses the fact that QTRCOURSE is hashed on instructors name. It is apparent that for simple queries of this type, none of the specialized database computers gives any significant increase in performance.



Fig. 7.3 Query Q2

LUCAS shows the best performance to a query Q2 of all the studied systems. If compared with INGRES, it is also 12 times faster in the worst case and 11 times faster in the best case. It is also 1.3 times faster than the second best design which is the much more complex DIRECT.

The poor performance of RAP, CASSM and CAFS is caused by their

inability to perform a Join operation. The host had to decompose query Q2 to a series of 22 subqueries.



Fig. 7.4 Query Q3

LUCAS is the fastest of all designs in executing Query 3. Also, it is 5 times faster than INGRES and 1.5 times faster than DIRECT.

7.5 INFLUENCE OF THE SIZE OF THE ASSOCIATIVE ARRAY

In previous sections it was assumed that the size of the Associative Array in LUCAS is 2k processors. We will now examine how the response times to queries Q1, Q2, and Q3 are influenced by an increase or decrease of this size.

Figure 7.5 shows the worst case response time for LUCAS with 0.5k, 1k, 2k, and 4k processors. The timing data were obtained by simple

modification of equations in Section 7.3. For example, in AAWORK for query Q2 in a 1k-case, instead of loading a cylinder 7 times, (7 times DAVAC), we load a half-cylinder 14 times (14 times DAVAC), etc.

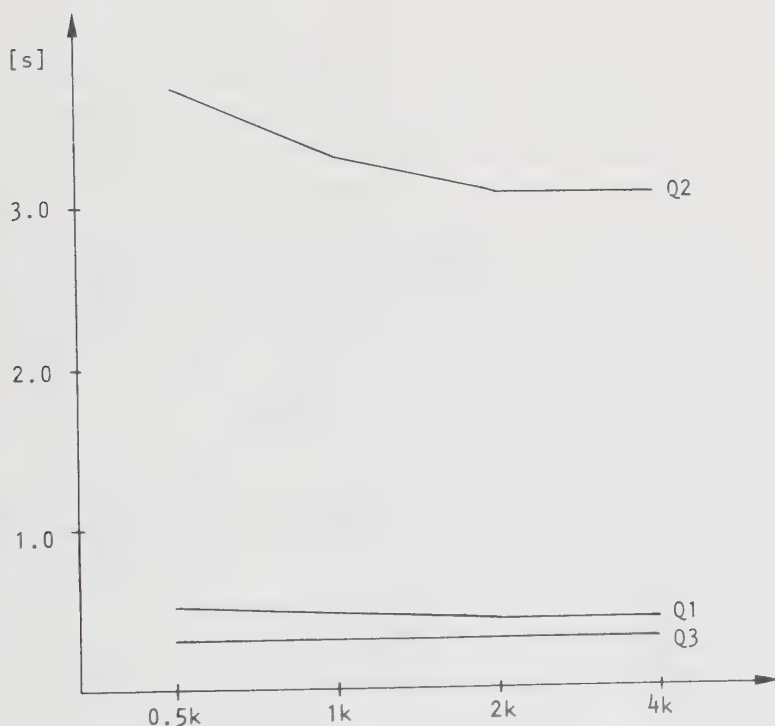


Fig. 7.5 Response times for different Array sizes

If we inspect Figure 7.5, we can make the following observations:

1) Even with an Associative Array four times smaller than the one that we have assumed in Section 7.4, the performance with respect to queries Q2 and Q3 is still very good. As a matter of fact it continues to be better than that of the other design.

2) The increase of the size of the Associative Array from 2k to 4k for query Q1, and from 0.5k to 4k for query Q3, does not lead to

an improvement of performance. The reason is that the interrogated relation in query Q1 can be stored in the Associative Array of the size $2k$, and the relation used in query Q3 can be stored in the Associative Array of the size $0.5k$. The computing potential of the excess processors in the Associative Array do not contribute in the computation.

3) There is no apparent improvement in response time to query Q2 if the number of processors in the Associative Array is increased from $2k$ to $4k$. The reason is not the idleness of processors - they are all utilized - but simply, that the Associative Array operates too fast compared to the I/O time. The Associative Array of the size $2k$ can store one cylinder load, the Array of $4k$ stores 2 cylinder loads. Because the main component of execution time of this query is the time to load data from the disks, a decrease of total processing time in the Associative Array in the $4k$ -case (as compared to a $2k$ -case) is negligible when compared to the I/O time.

An obvious question can be raised: Is there an optimal size of an Associative Array? We will discuss this question, in a more general setting, in the next chapter.

7.6 CONCLUSIONS

In this Chapter, we have compared the performance of a database computer containing an Associative Array with the performance of other database computers.

Our results indicate that in an environment of overhead-intensive queries, using an Associative Array does not give any advantage over a conventional database management system. On the contrary, the conventional system which uses standard techniques of hashing and indexing and thus can access only those pages on the disk which contain the result, in fact performs much faster.

The Associative Array must search through the whole relation indiscriminately. All its pages must be loaded, at the speed determined by the speed of disc, and even if they are then searched in an infinitesimally short time, the damage is already done and the execution time is too large.

The potential of the Associative Array can best be exploited in an environment of data-intensive multirelational queries, where its performance is better by an order of magnitude than that of a conventional system.

The choice of database and queries for the test was not ours. It does not by itself guarantee objectivity of our benchmarking experiment. But since LUCAS performed so remarkably well, the performance evaluation gives a clear indication that an Associative Array is a component of a database computer a viable possibility.

The performance of the Associative Array, if compared to INGRES and to other designs of database computers we have discussed, would probably be even much more impressive if we had used queries involving more than two relations. We have assumed an array width of 512 bytes, which is the current size of LUCAS. It was much more than was necessary and it would even allow for having more than two pages in the Array at the same time. We had no use for this feature when we evaluated queries Q1, Q2, and Q3. But in the case that a query involve more than two relations, or in the case that in the process of evaluation a series of algebraic operations creating intermediate relations must be performed, then the large width of the Array could be efficiently utilized. We have in Chapter 6 demonstrated a method of evaluating queries to a database residing in the Associative Array. A similar technique can be used when the size of relation exceeds the size of the Array. Since the width of the Associative Array costs less than its 'height' and Arrays with much larger memory width, an area in the Array could be used for permanently (or at least during a query session) storing contain information about

Chapter 8

EXTERNAL EVALUATION OF JOINS

In Chapter 7, we have evaluated the performance of a database computer containing an Associative Array of LUCAS type, in a real world situation. We have assumed a system consisting of an Associative Array connected to an Ampex 9200 disk drive. This system performed remarkably well in comparison to other designs.

If we analyze different components of the response times we can see that this time is determined largely by the properties of the disk drive: average access time, block read time, number of blocks/track, and number of tracks/cylinder. If, in our experiment, the Associative Array could process data ten times faster, it would have only a negligible influence on the total response time. Also, if the Associative Array were ten times slower (assuming constant I/O time), it would not have any significant influence on the execution time either. This observation indicates that the system is not properly balanced.

In this chapter, we will study a similar system as the one in Chapter 7, but this time we will assume that the properties of the disk are perfectly matched to the properties of the Associative Array. The rationale is that if we are going to design a database computer with such a powerful processing component as the Associative Array, then we will surely not want to rely the properties of a standard disk drive primarily aimed to be used with a sequential computer. We will instead modify the disk drive to achieve the highest system efficiency.

This 'ideal' Disk-Associative Array combination can be analyzed from

many points of view. We will restrict our investigation to a study of the cost efficiency of the execution of one of the most important operations in query processing, the join operation.

In conventional systems, there are two basic approaches to evaluate the join [Yao79].

1) Use of a nested loop algorithm. Tuples from the two argument relations are compared and matching tuples are output. This algorithm is very inefficient, since for two relations of the size N , it gives execution time proportional to $N*N$.

2) Use of sorting and merging, cf. Section 5.4, when the number of necessary comparisons of tuples is theoretically proportional to only $N*\log N$.

The second method is seemingly a much faster approach than the first one. The algorithms with the complexity growing as $N*\log N$ are in general considered to be good algorithms [Knuth73]. But in reality, in the case of very large relations where data are stored on the disk and must be brought into the CPU in pages, even the sort-merge algorithm is inherently slow due to large overheads. For example, DeWitt and Hawthorn in [DeWitt81] analyze execution of the join of two relations, with 30000 and 3000 tuples respectively, on a VAX 11/780 with an IBM 3330 disk drive, using sophisticated merge and sort algorithms. Examining the timing equation for e.g. the merge phase shows that 55 percent of the execution time is due to loading of pages to the main memory and only 45 percent is due to proper 'merging' in the CPU. Merging of two pages in the CPU is afflicted by further overheads and all this together gives inherently long total execution times.

This is an unsatisfactory situation. Fortunately, the performance of the join can be greatly improved by parallel join processors and many proposals can be found in the literature [Tong82]. We will not try to prove that our configuration is better, we simply investigate some consequences of our design choices.

We will determine the execution time and also whether there is an optimal size of the Associative Array for given statistical properties of relations.

We assume that the join will be performed on very large relations. According to [Oliver79], relations in very large databases typically contain 10^3 - 10^5 tuples. We must stress that what is considered to be a large relation is a function of time. Large relations in the future will be much larger than large relations today.

Since it is highly unlikely that in the reasonably near future it will be economically feasible to build Associative Arrays that are able to store large relations, we will assume that the size of the Array is less than one tenth of the cardinality of the larger of the two argument relations.

8.1 SYSTEM DESCRIPTION

We will assume the system configuration shown in Figure 8.1. The operand relations are stored on the disk. They are partitioned into pages of equal size. The disk will function as a large random access memory and the basic unit of data which is accessed is a page.

In commercial disk systems, the average access time is not much larger than a block read/write time. Since, as we will see, processing data in the Associative Array and returning the result to the host takes much longer time than loading data, we will assume that the time to locate pages of data can be overlapped with the processing time.

The Associative Array has the same capabilities as LUCAS. For example to compare one byte in a memory word with a byteslike in all memory words takes 48 clock cycles.

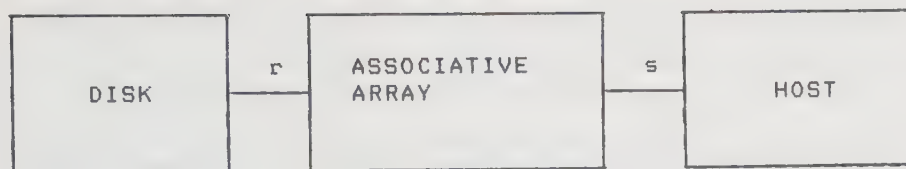


Fig 8.1 System architecture

This architecture is similar to the LUCAS system in Chapter 7. The main difference between these two systems are:

1) The data transfer rate, r , is now determined by the speed with which it can be input into the Associative Array - in Chapter 7 we only made sure that data coming from a disk could be loaded into the Array

2) The size of a page is determined by the size, c , of the Associative Array, a large Array means large pages, a small Array means small pages - in Chapter 7, the size of a page was equal to the size of a block on the disk.

First, we will determine r . We assume that the page is stored on the disk as a sequence of columns of bytes. When read from the disk, the first byte of the first tuple comes first and is loaded into the first I/O Register, then the first byte of the second tuple is loaded into the second I/O Register, and so on, until the first byte of the last, c -th tuple is loaded into the last I/O Register. After the whole column is loaded into the I/O Registers, it is shifted in 8 clock cycles into the proper bitaddress in the Associative Array. This procedure is repeated for each column. After every c bytes in the stream of bytes coming from the disk, which takes c clock cycles, there is a period of 8 clock cycles spent on shifting data from the I/O Registers into the memory words. During this period the Associative Array cannot receive any new data. For a large Associative Array, a necessary buffering of 8 bytes presents no practical difficulty. Thus, we assume that r is 1 byte/clock cycle. For a very small Associative Array, (and we will see that they must

be studied too), we will assume that we have more complex, 'double', I/O registers working in a complementary fashion, when one set is filled with data from the disk, the content of the other is shifted into the Array. Since we assume that r is 1 byte/clock cycle it also sets the limit on the minimal possible size of the Associative Array which is 8 processors.

Next, we must look at the value of s . It is the rate with which the bytes of the result tuples are returned to the host. We will assume that it is same as in Chapter 7, 10 clock cycles/byte (8 clock cycles for shifting + 2 for copying to the I/O Buffer Register). The difference in speed between loading and outputting of data is due to different modes of operation. Data are loaded as pages, but they are output as tuples which are selected according to their content.

8.2 ALGORITHM AND TIMING EQUATIONS

The algorithm which we will use is based on a tuple substitution algorithm [Wong76], where each tuple from one relation is compared with all tuples of the second and matching tuples are concatenated. The advantage of using parallel hardware is that in one operation, a tuple from one relation can be compared with a whole page of tuples of the second relation. In principle, a speed up equal to the parallelism in the hardware could be achieved.

We assume that the two operand relations are of the same cardinality, N tuples, and that the tuples have 2 attributes of b bytes each. The result relation (cf. Join1 in Chapter 3) has three attributes.

The execution proceeds in the following way, illustrated by Figure 8.2.

All pairs of pages of the operand relations are successively loaded into the Associative Array into areas I and II. For each pair, the values of joining attributes are compared in the way described in Section 5.3. Matching tuples are selected and serially output to the

host.

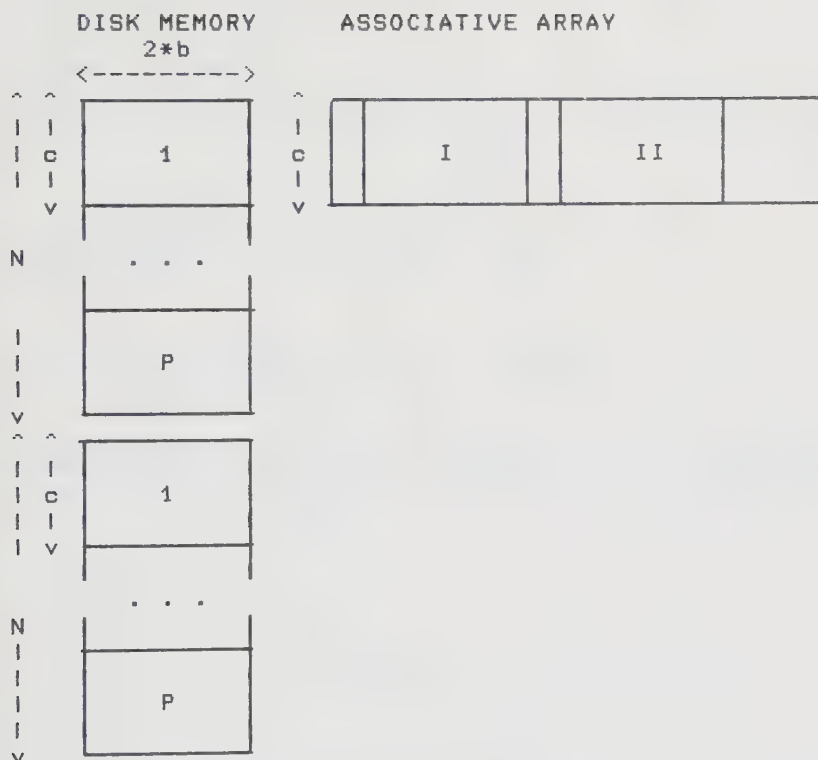


Fig. 8.2 Illustration of the execution of Join

We will characterize the content of the relations by a selectivity factor g , which can be defined as the ratio between the cardinality of the result of a join of two relations and the cardinality of their product. Intuitively, g is the probability that two tuples, randomly selected from the two operand relations, will match. We will assume that the values of joining attributes are evenly distributed.

Depending on the size of g , various proposals for the Join processors behave differently. There are two interesting situations which we will treat separately.

1) When the volume of data in the result relation is very much larger

than the volume of data in the operand relations. This case is assumed in a comparative study of different hardware approaches to Join processors by Tong and Yao [Tong82]. These authors assume a typical g being 0.5. It means that e.g. the Join of two relation of $10^3 - 10^5$ tuples each is a huge relation with the cardinality of $5 \cdot 10^6 - 5 \cdot 10^9$ tuples.

2) DeWitt and Hawthorn in an another study [DeWitt81] assume $g=0.0001$, 0.001 and 0.01 which gives considerably smaller result relations than in the case studied by Tong and Yao.

We will analyze the performance of the Associative Array with respect to both cases.

The execution time of the Join can be divided into three components:

$$T_{\text{Join}} = T_L + T_P + T_R$$

* T_L is the loading time. It is the time spent on loading data from the disk into the Associative Array.

* T_P is the processing time. It is the time spent on processing data in the Associative Array.

* T_R is the outputting time. It is the time spent sending concatenated result tuples to the host.

Loading time

The relations have $P=N/c$ pages each. There are P^2 pairs of pages, to load one page takes $2 \cdot c \cdot b$ clock cycles. Hence, the total loading time is

$$T_L = 2 \cdot P^2 \cdot c \cdot b$$

Outputting time

P^2 pairs of pages are compared. Each comparison of c tuples from the first relation with c tuples from the second produce in average

$c^2 \cdot g$ result tuples. One result tuple is $3 \cdot b$ bytes long, to output one byte takes 10 clock cycles, hence,

$$T_R = 30 \cdot P^2 \cdot b \cdot c^2 \cdot g$$

The overhead associated with selecting 'next selected tuple' for outputting is about 5 clock cycles/tuple and can be ignored.

Processing time

a) Large g

For a large selectivity factor ($g \gg 1/N$), if $g > 1/c$ then one page with a sample of c tuples contains as many different values of the joining attribute as the whole relation. Hence, to compare two pages of the operand relations only $1/g$ tuple comparisons must be made. Since the comparison takes 48 clock cycles/byte (cf. Section 5.3) we get,

$$T_{PG} = 48 \cdot P^2 \cdot b / g$$

b) Small g

For a small selectivity factor, we may assume that a sample of c tuples in one page contains c different values of the joining attribute. To compare a pair of pages, a tuple comparison must be made c times. Hence,

$$T_{Pg} = 48 \cdot P^2 \cdot b \cdot c$$

The timing equations are summarized in Table 8.1. We have substituted N/c for P .

T_L	T_R	T_{PG}	T_{Pg}
$2 \cdot N^2 \cdot b / c$	$30 \cdot N^2 \cdot b \cdot g$	$48 \cdot N^2 \cdot b / (c^2 \cdot g)$	$48 \cdot N^2 \cdot b / c$

Tab. 8.1 Timing equations for Join

8.3 DISCUSSION

The timing equations have the same general form as the timing equations of other designs of specialized Join procesors. The dependency on the cardinality of the operand relations and the dependency on the number of processors in the Associative Array can be expressed as

$$T = N^2 * A1, \text{ and } T = A2/c + A3$$

For example, the timing equations of the Two-dimensional Join Processor Array of *c* processors of Tong and Yao [Tong81] have the form

$$T = N^2 * B1, \text{ and } T = B2/c + B3/\text{sqrt}(c)$$

and the timing equations of a Join Processor of Menon and Hsiao [Menon81] have the form

$$T = N^2 * C1 + N * C2, \text{ and } T = C3/c + C4$$

We believe that - unless there will be some unexpected technological breakthrough - for very large relations, as long as data must be staged into Join processors, the timing equations for the total execution time of the Join on any hardware will always look approximately the same.

The timing equations in Table 8.1 show that the execution time decreases with the size of the Associative Array. But we can see that the total execution time contains a term which is independent of this size, the time for outputting selected tuples, T_R . Obviously, above some size of the Associative Array it will dominate, which means that further incresing the size of the Associative Array does not pay. We assume that a cut point of usefulness of the size of the Array occur when the time to output the result is 10 times larger than the sum of the time to load data and process them in the Array.

We will analyze the timing equations to see at which sizes of the Array the cut point occurs for different values of *g*.

Large g

To simplify our discussion, we will make the same assumption as the one made in [Tong82], i.e. $g=0.5$.

Hence, the timing equation is

$$T_{TJoinL} = N^2 * b * (96/c^2 + 15 + 2/c)$$

The following formula defines the cut point:

$$96/c^2 + 2/c = 1.5$$

We can see that the cut point occurs at $c=8$, and a further increase of the Array size above 8 processors gives only marginal improvement of performance.

We have assumed that to output a byte from the Associative Array takes ten times longer time than to load it (ten clock cycles versus one clock cycle) and we might suspect that that is the reason why the cut point value is so small. Let us assume that we could somehow speed up the time for outputting data from the Array, e.g. to 1 clock cycle/byte. The result would not be much better as the cut point is achieved already at 40 processors.

Since the idea of using the Associative Array assumes a large number of processors, our results clearly demonstrate that a bit-serially operating Associative Array is not a viable alternative as a candidate for a Join processor in the case of large values of g .

Small g

The first observation that we can make is that the loading time is 24 times shorter than the time spent on processing in the Associative Array and can thus be neglected.

After some simplification, we get the formula for the size of the cut point

$$c = 16/g$$

Table 8.2 gives the size of the cut point for different values of g .

g	c
0.01	$1.6 \cdot 10^3$
0.001	$1.6 \cdot 10^4$
0.0001	$1.6 \cdot 10^5$

Tab. 8.2 Cut points for small g

We can draw the conclusion that for small values of g , the large Associative Array is indeed suitable. Furthermore, we can with a large degree of confidence predict its optimum size.

We could see that in an analysis of the feasibility of the use of the Associative Array it was of outermost importance to make proper assumptions about the properties of data. For large g , i.e. any two tuples will match with a high probability, the Associative Array is obviously useless.

Tong and Yao [Tong82], analyzing the performance of different hardware solutions assume $g=0.5$ as being a worst case value. If an Associative Array was investigated according to their method the result would be very unfavorable for the simple reason that the Associative Array should certainly not be used in such a case. However, for small g , the result would indeed be different.

Chapter 9.

CONCLUSIONS AND FUTURE RESEARCH

The relational data model offers many advantages. To mention a few: It is very simple as seen by the user. It is symmetrical with respect to queries, so that there is no preferred format for a question to a database. It has also a very strong theoretical foundation.

There is only one problem which Chamberlin [Chamberlin76] calls the greatest open research question of the relational data model: can it be implemented efficiently? According to King, [King80] the answer, based on the experience with the IBMs System R, is a definite yes. King goes even one step further, claiming not only that relational systems can be implemented with reasonable performance but also that "exotic hardware (like associative memories, etc)" is not required. King certainly knows what he is talking about and is probably right. But there is another question: could this exotic hardware help, if it was available anyway?

The purpose of the research presented in this thesis was to study the applicability of an Associative Array in the design of a backend database computer whose function is to support a large database which utilizes the relational data model.

Our approach is exactly the one which was criticized by DeWitt and Hawthorn [DeWitt81] who give it the name "architecture directed" research.

... database machine designers usually begin by designing what they consider to be a good architecture which they feel will efficiently execute one or two database operations. Afterwards they develop the algorithms to support all the required database operations using the basic primitives of their architecture . . .

They advocate instead an approach where an architect of a database machine should start by first developing algorithms and then extract the primitive operations which are necessary for their efficient implementation. Only after these primitives are known and understood, should one attempt to design a machine.

We believe that DeWitt and Hawthorn confuse the different roles of research and development. How can an architect extract the primitive operations without (at least in the back of his head) knowing anything about the potential of different hardware solutions? Furthermore, how could it be possible to know anything about the potential of exotic hardware if there were nobody exploring it.

We have started by designing a model Associative Array, LUCAS, and our goal was to investigate its feasibility in database management applications. We consider our research being an exploratory basic research. We do not claim that we have proved that the Associative Array is a viable component of future database computers. But we claim that people who dismiss it, because they believe that its only capability is searching, are certainly mistaken.

9.1 CONCLUSIONS

The Associative Arrays do not enjoy widespread use at present, and so the research and development of algorithms utilizing this device is rather limited. In Chapter 5, we have described the implementation of algorithms for operations of relational algebra. The

implementation consisted mainly of transferring selected tuples of operand relations into the Comparand Register, comparing their values with the contents of the Associative Array, and logical processing of bitslices of information. The implementation of the Division operation is an example of utilizing the possibilities offered by the bitslice processing.

The execution time of an operation was typically proportional to the size of the tuple and to the cardinality of one of the argument relations. We have extrapolated our results and analyzed the performance of very large Associative Arrays. The absolute execution times, assuming a moderate 5 MHz clock frequency, were very good, and probably out of reach of present day sequential computers.

We have also discovered that the increase of performance gained by increasing the size of the Associative Array by e.g. a factor of 256 can be achieved on a sequential computer by doubling its processing speed. This is a rather negative result but it would be wrong to use it indiscriminately for refuting the Associative Arrays. Its impact can be softened by the following observations. We were comparing "ordinary" algorithms on the Associative Array with only one property of the best known sequential algorithms. For large volumes of data in the argument relations, it might be the case that overheads associated with 'preparing' data for sorting are much larger than the time for proper comparisons. Furthermore, sorting and merging of data has a disadvantage of consuming very large space in the memory of the sequential computer. We conclude this defence of an Associative Array by recalling that Date [Date82] claims that a typical software DBMSs average ten instructions per byte in evaluating a given selection condition on a given record.

In Chapter 6, we have demonstrated how algorithms from Chapter 5 can be used for evaluating complex queries entirely within the Associative Array. Having available a very large Associative Array than for a relatively stable set of queries (referring to the same relations over and over again), this method would be advantageous.

One very interesting avenue of research, which we have only very briefly discussed is the area of optimization of query evaluation. In our examples, the sequence of algebraic operations leading to the answer was stated ad hoc. But obviously, there can be more efficient ways to do it. What is the best strategy? How to identify bad strategies? How much from the research on optimization on sequential computer applies to evaluation according to our method. What "tricks" can be used to avoid the negative effects of the Join operation?

In Chapter 7, we have assumed a backend database computer consisting of an Associative Array connected to a disk. We have compared its performance with the performance of other well-documented designs of database computers and also with the performance of a conventional database management system. We have determined the response times to three types of queries. We could see that in an environment of overhead-intensive queries the Associative Array does not give any advantage over a conventional database management system. Comparison of response times to data-intensive or multirelational queries gave that the Associative Array is more than an order of magnitude better than the conventional system and also better than all the other studied designs. We do not want to overstate the importance of this finding, it was only one benchmark experiment, it can be the case that other tests could give different results. But still, since the Associative Array performed so remarkably well the obvious implications should not be ignored.

In Chapter 8, we went beyond the basic assumption of Chapters 5 and 6, which was that the Associative Array is larger than the operand relations. We dealt with the external evaluation of algebraic operations on very large relations. As an example we have chosen the Join operation. In a way, we were trying to answer the following question: Why is the performance of the Associative Array in Chapter 7 so good? We have identified two cases which depend on the content of the database: 1) large selectivity factor - where the Associative Array is useless, 2) small selectivity factor (which is

the case of queries Q2 and Q3 in Chapter 7) - where its use can be advantageous.

Throughout this thesis we have always seen LUCAS as a model of an Associative Array. Its basic characteristics are bit-serial mode of operation and a heavy use of transferring data between memory words and the Comparand Register. We will conclude this Section by some comments on LUCAS hardware. We have not discovered any wrong design decisions. Weakest point was the Control Unit. In a real system, the Address Processor must be much more powerful, and it must be able to maintain a much larger set of addresses. Also, the algorithms could be on the average 10 - 20 percent faster if we eliminated some loop-counting overheads, which are due to byte oriented organization of our algorithms. Although we have not implemented a hardware for counting the number of selected words, we know that it could be useful for implementing aggregate functions.

9.2 VLSI IMPLEMENTATION

No doubt, the Associative Array that we discuss in this thesis is very well suited for VLSI implementation. As part of a multi-project chip the logic of one processor (excluding memory) was implemented in CMOS/SOS in 1981.

Two of the major constraints in the design of VLSI components are the area of the chip and the number of connections between the logic on the chip and the outside world. We will now investigate what the consequences of integrating a large number of processors on one chip would be in terms of number of gate functions and number of pins per chip. (We assume that one gate function is implemented with four transistors.)

We assume that we want to implement only those facilities of LUCAS which are relevant for our investigation.

Two cases will be considered:

- * The consequences of using ordinary read/write memory chips as the memory words.
- * The consequences of including the memory on the chip.

Off-chip memory

Table 9.1 describes the number of pins needed on a chip comprising n processing elements. We assume b bits wide I/O data registers. In LUCAS, b is 8. Table 9.2 lists the function for different values of n for two different sizes of b .

Specification	pins
I/O data bus	b
I/O write	1
I/O data register address	$\log_2 n$
Chip select	1
Select First chain	2
Data in/out	n
Comparand data	1
Instruction code	5
Data control	2
Power, Ground, Clock	3
Sum: $b+15+n+\log_2 n$	

Tab. 8.1 The number of pins of an n processor chip

n	b=8	b=16
4	29	37
8	34	42
16	43	51
32	60	68
64	93	101
128	158	166
256	287	295

Tab. 8.2 The number of pins for different values of n

The first column with b=8 represents what is implemented on the current LUCAS. b=16 means improving the I/O rate by a factor of two.

The number of gate functions needed to implement the processors is totally dominated by the logic required to implement the ALUs. With 5 instruction bits, which are needed to specify the function, two flip-flops (T and R), a data bit from a Comparand register and a data bit from a memory word

$$G = 2 * 2^{(2+5+1+1)} = 2^{10}$$

gate functions are needed to implement the ALU of one processor element as a ROM. (This is an upper limit, since the number can be reduced considerably if a PLA is used instead of a ROM.)

In an n-processor chip, n*G gate functions are required. Table 8.3 lists n*G for different values of n.

n	n*G
4	2^{12}
8	2^{13}
16	2^{14}
32	2^{15}
64	2^{16}
128	2^{17}
256	2^{18}

Tab. 8.3 The number of memory cells required for an n-PE chip

Using these tables we can now choose a value of n that give values of gate count and number of pins within the limit of available technology.

Evidently, it is the number of pins and not the number of gate functions (memory cells) that limits the amount of processors which can realistically be placed on one chip. For example, a chip with 256 processors would have 287 pins which is a rather staggering number, but only 256k gate functions which it will be no technological problem to put on a chip in the near future.

If we assign 64 processors on a chip, we need only 93 pins and 64k gate functions and this is quite feasible with current VLSI technology.

We assumed that the memory modules were outside the chip. Suppose we want a memory word length of 4k bits. We could then use ordinary memory chips of 4k 8-bit words. Eight of these would be required to support one 64-processor chip of the above kind. An Associative Array with 512 processors, each with 4k bits of memory (72 chips), with some additional circuitry for I/O address decoding and buffering could easily be assembled on one circuit board.

On-chip memory

To provide addresses to bitslices in the memory, address pins are needed. We assume 2^m bits memory words, requiring m address bits. Furthermore, a write control signal is needed. Hence

$$b + 16 + \log_2 n + m$$

pins are needed (cf. Tab 8.1).

The number of gate functions needed for the memory modules would dominate the gate function count in such a chip. Assuming n processors with 2^m bits of memory each, $n \cdot 2^m$ gate functions are needed.

Table 8.4 lists the number of necessary pins and the number of gate functions for some values of n , assuming $m=12$.

n	gate functions	pins
4	2^{14}	38
8	2^{15}	39
16	2^{16}	40
32	2^{17}	41
64	2^{18}	42
128	2^{19}	43
256	2^{20}	44
512	2^{21}	45
1024	2^{22}	46

Tab 8.4 The number of pins of an n -processor chip

Clearly, it is the number of gate functions that puts the limit on what is implementable in this case.

Assume we can have 2^{21} (2 million) gate functions on a chip, and that the word length is chosen to be 2^{12} bits. Then 512 processors could be implemented on a single 45-pin chip. According to published figures [Patterson80], by the year 1991 about 11 million transistors

can be put on a chip. An Associative Array with thousands of processors would be easily built with these circuits on one circuit board.

REFERENCES

- Astrahan M.M., et al., "System R, A Relational Database Management System", IEEE Computer, Vol. 12, No. 5, May 1979
- Banerjee J., Baum R.I., Hsiao D.K., "Concepts and Capabilities of a Database Computer", ACM TODS, Vol. 3, No. 4, December 1978
- Banerjee J., Hsiao D.K., Kannan K., "DBC - A Database Computer for Very Large Databases", IEEE Trans. on Computers, Vol. C-28, No. 6, June 1979
- Batcher K.E., "STARAN parallel processor system hardware", Proc AFIPS 1974 National Computer Conf., AFIPS Press, 1974
- Bernstein P.A., Chiu D.W., "Using Semi-Joins to Solve Relational Queries", Journal of the Association for Computing Machinery, Vol. 28, No. 1, January 1981
- Berra B.P., Oliver E., "The Role of Associative Array Processors in Data Base Machine", IEEE Computer, Vol. 12, No. 3, March 1979
- Bray O.H., Freeman H.A., Data Base Computers, Lexington Books, Lexington, Mass., 1979
- Chamberlin D.D., "Relational Data-Base Management Systems", Computing Surveys, Vol. 8, No. 1, March 1976
- Codd E.F., "A relational model of data for large shared data banks", Comm. ACM, Vol. 13, No. 6, June 1970
- Codd E.F., "Relational Database: A Practical Foundation for Productivity", Comm. ACM, Vol. 25, No. 2, February 1982
- Date C.J., An Introduction to Database Systems, Addison-Wesley Publishing Company, Reading, Mass., 1981
- Date C.J., An Introduction to Database Systems - Volume II, Addison-Wesley Publishing Company, Mass., 1983
- DeFiore C., Berra P.B., "A Quantitative Analysis of the Utilization of Associative Memories in Data Base Management", IEEE Trans. on Computers, Vol. C-23, No. 2, February 1974
- DeWitt D. J., Friedland D., "Exploiting parallelism for the performance enhancement of non-numeric applications", Proc AFIPS 1982 National Computer Conf., AFIPS Press, 1982
- DeWitt D.J., Hawthorn P.B., "A Performance Evaluation of Database

Machine Architectures", Proc 7-th VLDB Conf., Cannes, September 1981

DeWitt D.J., "DIRECT - A multiprocessor organization for supporting relational database management systems", IEEE Trans. on Computers, Vol. C-28, No. 6, June 1979

Digby D. W., "A Search Memory for Many-to-Many Comparisons", IEEE Trans. on Computers, Vol. C-22, No. 8, August 1973

Epstein R., Hawthorn P., "Design Decissions for the Intelligent Database Machine", Proc AFIPS 1980 National Computer Conf., AFIPS Press, 1980

Fernström C., "Programming techniques on the LUCAS associative array computer", 1982 Int. Conf. on Parallel Processing, Bellaire, Michigan, August 1982

Fernström C., "The LUCAS Associative Array Processor and its Programming Environment", Doctoral Dissertation, Dept. of Computer Engineering, University of LUND, May 1983

Flynn M. F., "Some Computer Organizations and Their Effectiveness", IEEE Trans on Computers, Vol. C-21, No. 8, September 1972

Foster C. C., Content Addressable Parallel Processors, Van Nostrand Reinhold Company, New York, 1976

Gambino L.A., Boulis R.L., "STARAN Complex", Proc. 1975 Sagamore Computer Conf. on Parallel Processing, Sagamore, 1975

Hawthorn P. B., DeWitt D., "Performance Analysis of Alternative Database machine Architectures", IEEE Trans. on Software Engineering, Vol SE-8, No. 1, January 1982

Held G. D., Stonebraker M. R., and Wong E., "INGRES - A relational data base management system", Proc. AFIPS 1975 National Computer Conf., AFIPS Press, 1975

Hong Y.C, Su S.Y.W, "Associative Hardware and Software Techniques for Integrity Control", ACM TODS, Vol. 6., No. 3, September 1981

Hsiao D. K., "Data Base Computers", in Advances in Computers, Vol. 19, ed. Yovits M. C., Academic Press, Toronto, 1980

Hughes J., "Database processor extends mainframe functions to mini- or microcomputer systems", Computer Design, Vol. 21, No. 6, June 1982

King W.F.III, "Relational Database Systems: Where We Stand Today", Proc. IFIP Congress 1980

Knuth D., The Art of Computer Programming, Vol. 3, Addison-Wesley Publishing Company, Reading, Mass., 1973

Kruzela I. F., Svensson B. A., "The LUCAS Architecture and its Application to Relational Data Base Management", Proc of the 6-th Workshop on Computer Architecture for Non-Numerical Processing, INRIA, Hyeres, 1981

Kruzela I., "LUCAS Microprogramming Manual", Department of Computer Engineering, University of Lund, December 1980

Kuck D. J., The Structure of Computers and Computations - Volume 1, John Wiley and Sons, Toronto, 1978

Kung H. T., Lehman P. L., "Systolic (VLSI) Arrays for Relational Database Operations", Carnegie-Mellon University Report, CMU-80-114, 1980

Lamb S., "Recognition Memory Provides Basis for New Database Processor", MiniMicro Systems, Vol. 15, 1982

Lamb S., "An Add-in Recognition Memory for S-100 Bus Microcomputers - Part 1, Part 2, and Part 3", Computer Design, Vol.17, No. 8,9,10, 1978

Langdon G. G., "A note on associative processors for data management", ACM TODS, Vol. 3, No 2., June 1978

Lipovski G.J., Su S.Y.W., "Architectural Features of CASSM: A Context Addressed Segment Sequential Memory", Proc. 5th Annual Symposium on Computer Architecture, Palo Alto, April 1978

Maller V.A.J., "The Content Addressable File Store - CAFS", ICL Technical Journal, Vol. 1, No. 3., November 1979

McGee W. C., "Data Base Technology", IBM J. Res. Develop., Vol. 25, No. 5, September 1981

Menon M. J., Hsiao D. K., "Design and Analysis of A Relational Join Operation for VLSI", Proc 7-th VLDB Conf., Cannes, September 1981

Mick J., Brick J., Bit-Slice Microprocessor Design, McGraw-Hill Book Company, 1980

Moulder R., "An implementation of a data management system on an associative processor", Proc AFIPS 1973 National Computer Conf., AFIPS Press, 1973

Moulds P., "Dedicated machines take on data-base management", Electronic Design, Vol. 30, No. 6, June 1982

Oliver E. J., "RELACS, An Associative Computer Architecture to Support a Relational Data Model", Doctoral Dissertation, Syracuse University, 1979

Ozkarahan E.A., Schuster S.A., Sevcik K.C., "Performance

- evaluation of a Relational Associative Processor", ACM TODS, Vol. 2, No. 2, June 1977a
- Ozkarahan E.A., Sevcik K.C., "analysis of Architectural Features for Enhancing the Performance of a Database Machine", ACM TODS, Vol 2, No. 4, December 1977b
- Ozkarahan E.A., Schuster S.A., Smith K.C., "RAP - An Associative Processor for Data Base Management", Proc AFIPS 1975 National Computer Conf., AFIPS Press, 1975
- Pahrami B., "Associative Memories and Processors: An Overview and Selected Bibliography", Proc of the IEEE, Vol. 61, No. 6, June 1973
- Patterson D. A., Sequin C. H., "Design Consideration for Single-Chip Computers of the Future", IEEE Trans. on Computers, Vol. C-29, No. 2, February 1980
- Schuster S.A., Nguyen H.B., Ozkarahan E.A., Smith K.C., "RAP.2 - An Associative Processor for Data Bases", Proc. 5th Annual Symposium on Computer Architecture, Palo Alto, April 1978
- Shaw D. E., "Knowledge-Based Retrieval on A Relational Database Machine", Doctoral Dissertation, Stanford University, 1980
- Slotnik D. L., "Logic per track devices", in Advances in Computers, Vol. 10, ed. Alt F., Academic Press, Toronto 1970
- Song S. W., "On a High-Performance VLSI Solution to Database Problems", Carnegie-Mellon University Report, CMU-CS-81-142, August 1981
- Stone H. S., "Parallel Processing with the Perfect Shuffle", IEEE Transactions on Computers, Vol. C-20, February 1971
- Stonebraker M. R., Wong E., and Kreps P., "The Design and Implementation of INGRES", ACM TODS, Vol 1, No. 3, September 1976
- Su S. Y. W., Chang H., Copeland G., Fisher P., Lowenthal E., and Shuster S., "Database machines and Some issues on DBMS Standards", Proc AFIPS 1980 National Computer Conf., AFIPS Press, 1980
- Su S.Y.W., "Cellular Logic Devices: Concepts and Applications", IEEE Computer, Vol. 12, No. 3, March 1979
- Su S.Y.W., Lipovski G.J., "CASSM: A Cellular System for Very Large Data Bases", Proc. Int. Conf Very Large Databases, September 1975
- Svensson B., "LUCAS Processor Array - Design and Application", Doctoral Dissertation, Dept. of Computer Engineering, University of

LUND, April 1983

Thurber K. J., Large Scale Computer Architecture - Parallel and Associative Processors, Hyden Book Company Inc., New Jersey, 1976

Thurber K. J., Wald L. D., "Associative and Parallel Processors", Computing Surveys, Vol. 7, No. 4, December 1975

Tong F., Yao S. B., "Performance analysis of database join processors", Proc AFIPS 1982 National Computer Conf., AFIPS Press, 1982

Tong F., Yao B. S., "Design of a Two-Dimentional Join Processor Array", Proc of the 6-th Workshop on Computer Architecture for Non-Numerical Processing, INRIA, Hyeres, June 1981

Ullman J. D., Principles of Database Systems, Computer Science Press, Potomac, Maryland, 1980

Wong E., Youssefi K., "Decomposition - A Strategy for Query Processing", ACM TODS, Vol. 1, No. 3, September 1976

Yao S. B., "Optimization of Query Evaluation Algorithms", ACM TODS, Vol. 4, No. 2, June 1979

Yau S. S., Fung H. S., "Associative Processor Architecture - A Survey", Computing Surveys, Vol. 9, No. 1, March 1977

Zloof M. M., "Query By Example", Proc AFIPS 1975 National Computer Conf., AFIPS Press, 1975

Ivan Kruzela

AN ASSOCIATIVE ARRAY PROCESSOR – Supporting a Relational Algebra

The purpose of the research presented in this thesis is to study the applicability of an Associative Array in the design of a backend database computer whose function is to support a large database which utilizes the relational data model.

As part of the study an Associative Array Processor, LUCAS, has been implemented. It comprises 128 bit-serial processors working in SIMD mode.

Two related theses report further on design issues, programming and application studies:

Christer Fernström: The LUCAS Associative Array Processor and its Programming Environment, PhD thesis 1983.

Bertil Svensson: LUCAS Processor Array – Design and Applications, PhD thesis 1983